```
import numpy as np
from PIL import Image
from torch.utils import data
import glob
import cv2
import utils

from torchvision import transforms as pth_transforms

import main_run

parser = main_run.get_args_parser()
args = parser.parse_args()

class CXR_Dataset(data.Dataset):
  'Characterizes a dataset for PyTorch'

  def __init__(self, data_dir, transforms=None, mode='train', labeled=False):
    'Initialization'
    self.dim = (256, 256)
    self.n_classes = 1
    self.transforms = transforms
    self.mode = mode
    self.labeled = labeled

    if args.total_folds == 0:
```

```
    self.total_folds = ['labeled']
    self.pseudo_folds = None
elif args.total_folds == 1:
    self.total_folds = ['labeled']
    self.pseudo_folds = ['fold_0']
elif args.total_folds == 2:
    self.total_folds = ['labeled']
    self.pseudo_folds = ['fold_0', 'fold_1']
elif args.total_folds == 3:
    self.total_folds = ['labeled']
    self.pseudo_folds = ['fold_0', 'fold_1', 'fold_2']

self.test_fold = ['test']

self.total_images = {}

# PNG and JPG image lists
if self.mode == 'train':
    if self.labeled == True:
        for fold in self.total_folds:   # Fix total folds to 1 during self-training
            png_lists = glob.glob(data_dir + '{}/'.format(fold) + '**/*.png', recursive=True)
            jpg_lists = glob.glob(data_dir + '{}/'.format(fold) + '**/*.jpg', recursive=True)
            for png in png_lists:
                self.total_images[png] = 'label'
            for jpg in jpg_lists:
                self.total_images[jpg] = 'label'
```

```
        if self.labeled == False:
            for p_fold in self.pseudo_folds:
                p_png_lists = glob.glob(data_dir + '{}/'.format(p_fold) + '**/*.png', recursive=True)
                p_jpg_lists = glob.glob(data_dir + '{}/'.format(p_fold) + '**/*.jpg', recursive=True)
                for p_png in p_png_lists:
                    self.total_images[p_png] = 'pseudo'
                for p_jpg in p_jpg_lists:
                    self.total_images[p_jpg] = 'pseudo'

    elif self.mode == 'test':
        for t_fold in self.test_fold:
            png_lists = glob.glob(data_dir + '{}/'.format(t_fold) + '**/*.png', recursive=True)
            jpg_lists = glob.glob(data_dir + '{}/'.format(t_fold) + '**/*.jpg', recursive=True)
            for png in png_lists:
                self.total_images[png] = 'label'
            for jpg in jpg_lists:
                self.total_images[jpg] = 'label'

    self.total_images_list = sorted(self.total_images.keys())
    self.selected_images = self.total_images_list

    print('A total of %d image data were generated.' % len(self.selected_images))
    self.n_data = len(self.selected_images)

def __len__(self):
```

```
    'Denotes the total number of samples'
    return self.n_data

def __getitem__(self, index):
    'Generates one sample of data'

    img_path = self.total_images_list[index]
    image = cv2.imread(img_path, 1)

    image = cv2.resize(image, dsize=(256, 256), interpolation=cv2.INTER_LINEAR)

    image = Image.fromarray(image)

    # Apply DINO augmentation
    if not self.transforms == None:
        images = self.transforms(image)

    # Apply NO augmentation
    elif self.transforms == None:
        image = pth_transforms.Compose(
            [utils.GaussianBlurInference(),
             pth_transforms.ToTensor()])(image)

    idx = img_path

    # Make label
```

```
        if self.labeled == True:
            if 'Normal' in idx:
                label = 0
            elif 'Tuberculosis' in idx:
                label = 1
        else:
            # Label without meaning
            label = 9999

        if self.mode == 'train':
            return images, label
        else:
            return image, label

# coding=utf-8
from __future__ import absolute_import, division, print_function

import logging
import argparse
import os

import numpy as np

import torch

from tqdm import tqdm
```

```
from torch.utils.tensorboard import SummaryWriter
from sklearn.metrics import classification_report
from sklearn.metrics import roc_auc_score
import torch.nn as nn
import CXR_dataset
from torch.utils.data import DataLoader, SequentialSampler
import utils
from vision_transformer import DINOHead, CLS_head
from torchvision import models as torchvision_models

import vision_transformer as vit_o
from main_run import get_args_parser

parser = argparse.ArgumentParser('DINO', parents=[get_args_parser()])
args = parser.parse_args()

logger = logging.getLogger(__name__)

class AverageMeter(object):
  """Computes and stores the average and current value"""
  def __init__(self):
    self.reset()

  def reset(self):
    self.val = 0
    self.avg = 0
```

```
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count


def simple_accuracy(preds, labels):
    return (preds == labels).mean()


def save_model(args, iteration, model, optimizer, scheduler, best_auc):
    model_to_save = model.module if hasattr(model, 'module') else model
    model_checkpoint = os.path.join(args.output_dir, "%s_checkpoint.bin" % args.name)
    torch.save({'iteration': iteration, 'model_state_dict': model_to_save.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'best_auc': best_auc}, model_checkpoint)
    logger.info("Saved model checkpoint to [DIR: %s]", args.output_dir)

def save_model_latest(args, iteration, model, optimizer, scheduler, best_auc):
    model_to_save = model.module if hasattr(model, 'module') else model
```

```
    model_checkpoint = os.path.join(args.output_dir, "%s_checkpoint_latest.bin" %
args.name)
    torch.save({'iteration': iteration, 'model_state_dict': model_to_save.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'scheduler_state_dict': scheduler.state_dict(),
            'best_auc': best_auc}, model_checkpoint)
    logger.info("Saved latest model checkpoint to [DIR: %s]", args.output_dir)

def count_parameters(model):
    params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    return params/1000000

def valid(args, model, writer, test_loader):
    # Validation!
    eval_losses = AverageMeter()

    logger.info("***** Running Validation *****")
    logger.info("  Num steps = %d", len(test_loader))
    logger.info("  Batch size = %d", 1)

    model.eval()

    pred, true = [], []
    y_score_1 = []

    epoch_iterator = tqdm(test_loader,
```

```
                desc="Validating... (loss=X.X)",
                bar_format="{l_bar}{r_bar}",
                dynamic_ncols=True,
                disable=args.local_rank not in [-1, 0])

loss_fct = nn.BCEWithLogitsLoss()

for step, batch in enumerate(epoch_iterator):
   x, y = batch
   x = x.to(args.device)
   y = y.to(args.device).float()

   with torch.no_grad():
      output = model(x)
      labels = y

      loss = loss_fct(output.view(-1), labels.view(-1))
      eval_losses.update(loss.item())

      output = torch.sigmoid(output + 0.0)
      prob_np = output.detach().cpu().numpy()
      preds = np.round(prob_np)

      for x in range(len(labels)):
         true.append(np.asarray(labels.cpu())[x])
      for x in range(len(labels)):
```

```
            pred.append(np.asarray(preds)[x])

        # Calculate score for AUC
        for x in range(len(labels)):
            y_sc = prob_np[x]
            y_score_1.append(y_sc)

    epoch_iterator.set_description("Validating... (loss=%2.5f)" % eval_losses.val)

y_score = np.array(y_score_1)

auc = roc_auc_score(true, y_score)

logger.info("\n")
logger.info("External validation Results")
logger.info("Valid Loss: %2.5f" % eval_losses.avg)
logger.info("External validation AUC (TB): %2.5f" % auc)

logger.info(
    classification_report(y_true=true, y_pred=pred, target_names=['Normal',
'Tuberculosis'],
                          digits=4, labels=list(range(2))))

writer.add_scalar("test/loss", scalar_value=eval_losses.avg)
return eval_losses.avg, auc
```

```
def evaluate(args, model):
    """ Train the model """
    if args.local_rank in [-1, 0]:
        os.makedirs(args.output_dir, exist_ok=True)
        writer = SummaryWriter(log_dir=os.path.join("logs", args.name))

    # Prepare dataset
    testset = CXR_dataset.CXR_Dataset(args.data_path, transforms=None, mode='test',
labeled=True)

    test_sampler = SequentialSampler(testset)
    test_loader = DataLoader(testset,
                             sampler=test_sampler,
                             batch_size=1,
                             num_workers=8,
                             pin_memory=True) if testset is not None else None

    # Load weights
    state_dict = torch.load(args.pretrained_dir, map_location="cpu")
    args.checkpoint_key = 'student'
    print("Take key {} in provided checkpoint dict".format(args.checkpoint_key))
    state_dict = state_dict[args.checkpoint_key]
    # remove `module.` prefix
    state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
    msg_s = model.load_state_dict(state_dict, strict=False)
```

```
    print('Weights found at {} and loaded with msg: {}'.format('CheXpert and pre-training', msg_s))

    loss, auc = valid(args, model, writer, test_loader)

    logger.info("Best AUC: \t%f" % auc)
    logger.info("End Validation!")


def main():
    # Setup CUDA, GPU & distributed training
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    args.n_gpu = torch.cuda.device_count()
    args.device = device

    # Setup logging
    logging.basicConfig(format='%(asctime)s - %(levelname)s - %(name)s - %(message)s',
                        datefmt='%m/%d/%Y %H:%M:%S',
                        level=logging.INFO if args.local_rank in [-1, 0] else logging.WARN)

    # Define model and load weights
    if 'vit' in args.arch:
        model = vit_o.__dict__[args.arch](patch_size=args.patch_size)
        embed_dim = model.embed_dim
        inter_dim = 384
```

```
    elif args.arch in torchvision_models.__dict__.keys():
        model = torchvision_models.__dict__[args.arch]()

        if 'resne' in args.arch:
            embed_dim = model.fc.weight.shape[1]
            inter_dim = 2048
        elif 'densenet' in args.arch:
            embed_dim = model.classifier.weight.shape[1]
            inter_dim = 1920
        elif 'eff' in args.arch:
            embed_dim = model.classifier[1].weight.shape[1]
            inter_dim = 1792

    model = utils.MultiCropWrapper(
        model,
        DINOHead(embed_dim, args.out_dim, args.use_bn_in_head), CLS_head(inter_dim,
256, 1), args)

    model = model.cuda()

    # Training
    evaluate(args, model)



if __name__ == "__main__":
    main()
```

```
# Copyright (c) Facebook, Inc. and its affiliates.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
import torch
from torchvision.models.resnet import resnet50

import vision_transformer as vit_o

dependencies = ["torch", "torchvision"]


def dino_vits16(pretrained=True, **kwargs):
    """
    ViT-Small/16x16 pre-trained with DINO.
    Achieves 74.5% top-1 accuracy on ImageNet with k-NN classification.
```

```
    """
    model = vit_o.__dict__["vit_small"](patch_size=16, num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
url="https://dl.fbaipublicfiles.com/dino/dino_deitsmall16_pretrain/dino_deitsmall16_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
    return model


def dino_vits8(pretrained=True, **kwargs):
    """
    ViT-Small/8x8 pre-trained with DINO.
    Achieves 78.3% top-1 accuracy on ImageNet with k-NN classification.
    """
    model = vit_o.__dict__["vit_small"](patch_size=8, num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
url="https://dl.fbaipublicfiles.com/dino/dino_deitsmall8_pretrain/dino_deitsmall8_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
```

```
    return model


def dino_vitb16(pretrained=True, **kwargs):
    """
    ViT-Base/16x16 pre-trained with DINO.
    Achieves 76.1% top-1 accuracy on ImageNet with k-NN classification.
    """
    model = vit_o.__dict__["vit_base"](patch_size=16, num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
url="https://dl.fbaipublicfiles.com/dino/dino_vitbase16_pretrain/dino_vitbase16_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
    return model


def dino_vitb8(pretrained=True, **kwargs):
    """
    ViT-Base/8x8 pre-trained with DINO.
    Achieves 77.4% top-1 accuracy on ImageNet with k-NN classification.
    """
    model = vit_o.__dict__["vit_base"](patch_size=8, num_classes=0, **kwargs)
    if pretrained:
```

```
    state_dict = torch.hub.load_state_dict_from_url(
url="https://dl.fbaipublicfiles.com/dino/dino_vitbase8_pretrain/dino_vitbase8_pretrain.pth",
        map_location="cpu",
    )
    model.load_state_dict(state_dict, strict=True)
  return model


def dino_resnet50(pretrained=True, **kwargs):
  """
  ResNet-50 pre-trained with DINO.
  Achieves 75.3% top-1 accuracy on ImageNet linear evaluation benchmark (requires to train `fc`).
  """
  model = resnet50(pretrained=False, **kwargs)
  model.fc = torch.nn.Identity()
  if pretrained:
    state_dict = torch.hub.load_state_dict_from_url(
url="https://dl.fbaipublicfiles.com/dino/dino_resnet50_pretrain/dino_resnet50_pretrain.pth",
        map_location="cpu",
    )
    model.load_state_dict(state_dict, strict=False)
  return model
```

```
def dino_xcit_small_12_p16(pretrained=True, **kwargs):
    """
    XCiT-Small-12/16 pre-trained with DINO.
    """
    model = torch.hub.load('facebookresearch/xcit', "xcit_small_12_p16", num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
            url="https://dl.fbaipublicfiles.com/dino/dino_xcit_small_12_p16_pretrain/dino_xcit_small_12_p16_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
    return model


def dino_xcit_small_12_p8(pretrained=True, **kwargs):
    """
    XCiT-Small-12/8 pre-trained with DINO.
    """
    model = torch.hub.load('facebookresearch/xcit', "xcit_small_12_p8", num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
```

```
url="https://dl.fbaipublicfiles.com/dino/dino_xcit_small_12_p8_pretrain/dino_xcit_small_12_p8_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
    return model


def dino_xcit_medium_24_p16(pretrained=True, **kwargs):
    """
    XCiT-Medium-24/16 pre-trained with DINO.
    """
    model = torch.hub.load('facebookresearch/xcit', "xcit_medium_24_p16", num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
url="https://dl.fbaipublicfiles.com/dino/dino_xcit_medium_24_p16_pretrain/dino_xcit_medium_24_p16_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
    return model


def dino_xcit_medium_24_p8(pretrained=True, **kwargs):
```

```
    """
    XCiT-Medium-24/8 pre-trained with DINO.
    """
    model = torch.hub.load('facebookresearch/xcit', "xcit_medium_24_p8", num_classes=0, **kwargs)
    if pretrained:
        state_dict = torch.hub.load_state_dict_from_url(
            url="https://dl.fbaipublicfiles.com/dino/dino_xcit_medium_24_p8_pretrain/dino_xcit_medium_24_p8_pretrain.pth",
            map_location="cpu",
        )
        model.load_state_dict(state_dict, strict=True)
    return model


import argparse
import os
import sys
import datetime
import time
import math
import json
from pathlib import Path
import csv

import numpy as np
from PIL import Image
```

```
import torch
import torch.nn as nn
import torch.distributed as dist
import torch.backends.cudnn as cudnn
import torch.nn.functional as F
from torchvision import transforms
from torchvision import models as torchvision_models

import utils
import vision_transformer as vit_o
from vision_transformer import DINOHead, CLS_head

torchvision_archs = sorted(name for name in torchvision_models.__dict__
  if name.islower() and not name.startswith("__")
  and callable(torchvision_models.__dict__[name]))

def get_args_parser():
  parser = argparse.ArgumentParser('DINO', add_help=False)

  # Model parameters
  parser.add_argument("--name", required=True,
            help="Name of this run. Used for monitoring.")
  parser.add_argument('--use_original', default=True, type=bool,
            help="Whether to use original ViT (DINO, not ours) model or not")
  parser.add_argument('--fine_tune', default=False, type=bool,
            help="Fine-tune network or not")
```

```
    parser.add_argument('--arch', default='vit_small', type=str,
        choices=['vit_tiny', 'vit_small', 'vit_base'] + torchvision_archs + torch.hub.list("facebookresearch/xcit"),
        help="""Name of architecture to train. For quick experiments with ViTs,
        we recommend using vit_small.""")
    parser.add_argument('--patch_size', default=8, type=int, help="""Size in pixels    # Original - 8, ours - 16
        of input square patches - default 16 (for 16x16 patches). Using smaller
        values leads to better performance but requires more memory. Applies only
        for ViTs (vit_tiny, vit_small and vit_base). If <16, we recommend disabling
        mixed precision training (--use_fp16 false) to avoid unstabilities.""")
    parser.add_argument('--out_dim', default=65536, type=int, help="""Dimensionality of
        the DINO head output. For complex and large datasets large values (like 65k) work well.""")
    parser.add_argument('--norm_last_layer', default=True, type=utils.bool_flag,
        help="""Whether or not to weight normalize the last layer of the DINO head.
        Not normalizing leads to better performance but can make the training unstable.
        In our experiments, we typically set this paramater to False with vit_small and True with vit_base.""")
    parser.add_argument('--momentum_teacher', default=0.9995, type=float, help="""Ba``se EMA
        parameter for teacher update. The value is increased to 1 during training with cosine schedule.
        We recommend setting a higher value with small batches: for example use 0.9995 with batch size of 256.""")
    parser.add_argument('--use_bn_in_head', default=False, type=utils.bool_flag,
        help="Whether to use batch normalizations in projection head (Default: False)")
```

```
    # Temperature teacher parameters
    parser.add_argument('--warmup_teacher_temp', default=0.04, type=float,
        help="""Initial value for the teacher temperature: 0.04 works well in most cases.
        Try decreasing it if the training loss does not decrease.""")
    parser.add_argument('--teacher_temp', default=0.04, type=float, help="""Final value (after linear warmup)
        of the teacher temperature. For most experiments, anything above 0.07 is unstable. We recommend
        starting with the default value of 0.04 and increase this slightly if needed.""")
    parser.add_argument('--warmup_teacher_temp_epochs', default=0, type=int,
        help='Number of warmup epochs for the teacher temperature (Default: 30).')

    # Training/Optimization parameters
    parser.add_argument('--use_fp16', type=utils.bool_flag, default=True, help="""Whether or not
        to use half precision for training. Improves training time and memory requirements,
        but can provoke instability and slight decay of performance. We recommend disabling
        mixed precision if the loss is unstable, if reducing the patch size or if training with bigger ViTs.""")
    parser.add_argument('--weight_decay', type=float, default=0.01, help="""Initial value of the
        weight decay. With ViT, a smaller value at the beginning of training works well.""")
    parser.add_argument('--weight_decay_end', type=float, default=0.01, help="""Final value of the
        weight decay. We use a cosine schedule for WD and using a larger decay by
        the end of training improves performance for ViTs.""")
    parser.add_argument('--clip_grad', type=float, default=3.0, help="""Maximal parameter
        gradient norm if using gradient clipping. Clipping with norm .3 ~ 1.0 can
```

```
        help optimization for larger ViT architectures. 0 for disabling.""")
    parser.add_argument('--batch_size_per_gpu', default=8, type=int,
        help='Per-GPU batch-size : number of distinct images loaded on one GPU.')
    parser.add_argument('--epochs', default=5, type=int, help='Number of epochs of
training.')
    parser.add_argument('--ssl_epoch', default=5, type=int, help='Number of epochs of self-
supervised learning.')
    parser.add_argument('--freeze_last_layer', default=1, type=int, help="""Number of
epochs
        during which we keep the output layer fixed. Typically doing so during
        the first epoch helps training. Try increasing this value if the loss does not decrease.""")
    parser.add_argument("--lr", default=0.00005, type=float, help="""Learning rate at the end
of
        linear warmup (highest LR used during training). The learning rate is linearly scaled
        with the batch size, and specified here for a reference batch size of 256.""")
    parser.add_argument("--warmup_epochs", default=1, type=int,
        help="Number of epochs for the linear learning-rate warm up.")
    parser.add_argument('--min_lr', type=float, default=1e-6, help="""Target LR at the
        end of optimization. We use a cosine LR schedule with linear warmup.""")
    parser.add_argument('--optimizer', default='adamw', type=str,
        choices=['adamw', 'sgd', 'lars'], help="""Type of optimizer. We recommend using adamw
with ViTs.""")
    parser.add_argument('--drop_path_rate', type=float, default=0.1, help="stochastic depth
rate")


    # Multi-crop parameters
    parser.add_argument('--global_crops_scale', type=float, nargs='+', default=(0.75, 1.),
```

```
            help="""Scale range of the cropped save before resizing, relatively to the origin save.
            Used for large global view cropping. When disabling multi-crop (--local_crops_number 0), we
            recommand using a wider range of scale ("--global_crops_scale 0.14 1." for example)""")
    parser.add_argument('--local_crops_number', type=int, default=8, help="""Number of small
            local views to generate. Set this parameter to 0 to disable multi-crop training.
            When disabling multi-crop we recommend to use "--global_crops_scale 0.14 1." """)
    parser.add_argument('--local_crops_scale', type=float, nargs='+', default=(0.2, 0.6),
            help="""Scale range of the cropped save before resizing, relatively to the origin save.
        Used for small local view cropping of multi-crop.""")


    # Misc
    # PATH TO DATA
    parser.add_argument('--data_path', default='/PATH/DATA/', type=str,
      help='Please specify the directory to your data')
    # PATH TO SAVE MODEL WEIGHTS
    parser.add_argument('--output_dir', default="/PATH/SAVE/", type=str,
            help='Path to save logs and checkpoints.')
    parser.add_argument('--saveckp_freq', default=20, type=int, help='Save checkpoint every x epochs.')
    parser.add_argument('--seed', default=0, type=int, help='Random seed.')
    parser.add_argument('--num_workers', default=10, type=int, help='Number of data loading workers per GPU.')
    parser.add_argument("--dist_url", default="env://", type=str, help="""url used to set up
```

```
    distributed training; see https://pytorch.org/docs/stable/distributed.html""")
  parser.add_argument("--local_rank", default=0, type=int, help="Please ignore and do not set this argument.")

  parser.add_argument('--total_folds', type=int, default=0,
                      help='Total folds to include')
  parser.add_argument("--pretrained_dir", type=str, default='./pretrained_weights/pertrain.ckpt',
                      help="Where to search for pretrained ViT models on CheXpert features.")
  parser.add_argument("--checkpoint_key", default="each", type=str,
                      help='Key to use in the checkpoint (example: "student", "teacher")')

  parser.add_argument('--lam', type=float, default=0.5,
                      help="Loss scaling")
  parser.add_argument("--correct", default=500, type=int,
                      help="correction steps")

  parser.add_argument('--alpha', default=True, type=bool,
                      help="Use alpha weight")

  return parser

import CXR_dataset

def train_dino(args):
  utils.init_distributed_mode(args)
  utils.fix_random_seeds(args.seed)
```

```
    print("git:\n  {}\n".format(utils.get_sha()))
    print("\n".join("%s: %s" % (k, str(v)) for k, v in sorted(dict(vars(args)).items())))
    cudnn.benchmark = True

    # ============ preparing data ... ============
    transform = DataAugmentationDINO(
        args.global_crops_scale,
        args.local_crops_scale,
        args.local_crops_number,
    )
    dataset = CXR_dataset.CXR_Dataset(args.data_path, transforms=transform,
mode='train', labeled=False)
    sampler = torch.utils.data.DistributedSampler(dataset, shuffle=True)
    data_loader = torch.utils.data.DataLoader(
        dataset,
        sampler=sampler,
        batch_size=args.batch_size_per_gpu,
        num_workers=args.num_workers,
        pin_memory=True,
        drop_last=True,
    )

    l_dataset = CXR_dataset.CXR_Dataset(args.data_path, transforms=transform,
mode='train', labeled=True)
    l_sampler = torch.utils.data.DistributedSampler(l_dataset, shuffle = True)
    l_data_loader = torch.utils.data.DataLoader(
        l_dataset,
```

```
        sampler=l_sampler,
        batch_size=args.batch_size_per_gpu,
        num_workers=args.num_workers,
        pin_memory=True,
        drop_last=True,
    )

    print(f"Unlabeled Data loaded: there are {len(dataset)} images.")
    print(f"Labeled Data loaded: there are {len(l_dataset)} images.")

    # ============ building student and teacher networks ... ============
    # we changed the name DeiT-S for ViT-S to avoid confusions
    # if the network is a Vision Transformer (i.e. vit_tiny, vit_small, vit_base)
    if args.arch in vit_o.__dict__.keys():
        student = vit_o.__dict__[args.arch](
            patch_size=args.patch_size,
            drop_path_rate=args.drop_path_rate,  # stochastic depth
        )
        teacher = vit_o.__dict__[args.arch](patch_size=args.patch_size)
        embed_dim = student.embed_dim
        inter_dim = 384

    elif args.arch == 'deit':
        student = torch.hub.load('facebookresearch/deit:main', 'deit_base_patch16_384',
pretrained=True,
                                 drop_path_rate=args.drop_path_rate)
```

```
    teacher = torch.hub.load('facebookresearch/deit:main', 'deit_base_patch16_384',
pretrained=True)
    embed_dim = student.embed_dim


  # if the network is a XCiT
  elif args.arch in torch.hub.list("facebookresearch/xcit"):
    student = torch.hub.load('facebookresearch/xcit', args.arch,
                             pretrained=False, drop_path_rate=args.drop_path_rate)
    teacher = torch.hub.load('facebookresearch/xcit', args.arch, pretrained=False)
    embed_dim = student.embed_dim
  # otherwise, we check if the architecture is in torchvision models
  elif args.arch in torchvision_models.__dict__.keys():
    student = torchvision_models.__dict__[args.arch]()
    teacher = torchvision_models.__dict__[args.arch]()


    if 'resne' in args.arch:
      embed_dim = student.fc.weight.shape[1]
      inter_dim = 2048
    elif 'densenet' in args.arch:
      embed_dim = student.classifier.weight.shape[1]
      inter_dim = 1920
    elif 'eff' in args.arch:
      embed_dim = student.classifier[1].weight.shape[1]
      inter_dim = 1792
  else:
    print(f"Unknow architecture: {args.arch}")
```

```
  # multi-crop wrapper handles forward with inputs of different resolutions
  student = utils.MultiCropWrapper(student, DINOHead(
    embed_dim,
    args.out_dim,
    use_bn=args.use_bn_in_head,
    norm_last_layer=args.norm_last_layer,
  ), CLS_head(inter_dim, 256, 1), args)
  teacher = utils.MultiCropWrapper(
    teacher,
    DINOHead(embed_dim, args.out_dim, args.use_bn_in_head), CLS_head(inter_dim,
256, 1), args)


  # Load weights
  state_dict = torch.load(args.pretrained_dir, map_location="cpu")
  print("Take key {} in provided checkpoint dict".format(args.checkpoint_key))
  if args.checkpoint_key == 'each':
    if args.total_folds == 1:
      state_dict_t = state_dict['student']
      state_dict_s = state_dict['student']
    else:
      state_dict_t = state_dict['teacher']
      state_dict_s = state_dict['student']
  else:
    if args.total_folds == 1:
      state_dict_t = state_dict['student']
```

```
        state_dict_s = state_dict['student']
    else:
        state_dict_t = state_dict[args.checkpoint_key]
        state_dict_s = state_dict[args.checkpoint_key]
  # remove `module.` prefix
  state_dict_t = {k.replace("module.", ""): v for k, v in state_dict_t.items()}
  state_dict_s = {k.replace("module.", ""): v for k, v in state_dict_s.items()}
  # remove `backbone.` prefix induced by multicrop wrapper
  # state_dict = {k.replace(".backbone.", "___"): v for k, v in state_dict.items()}
  # state_dict = {k.replace("backbone.", ""): v for k, v in state_dict.items()}
  # state_dict = {k.replace("___", ".backbone."): v for k, v in state_dict.items()}
  msg_t = teacher.load_state_dict(state_dict_t, strict=False)
  print('(Teacher) Pretrained weights found at {} and loaded with msg: {}'.format('Pre-
training', msg_t))
  msg_s = student.load_state_dict(state_dict_s, strict=False)
  print('(Student) Pretrained weights found at {} and loaded with msg: {}'.format('Pre-
training', msg_s))

  # move networks to gpu
  student, teacher = student.cuda(), teacher.cuda()
  # synchronize batch norms (if any)
  if utils.has_batchnorms(student):
    student = nn.SyncBatchNorm.convert_sync_batchnorm(student)
    teacher = nn.SyncBatchNorm.convert_sync_batchnorm(teacher)

    # we need DDP wrapper to have synchro batch norms working...
```

```
        teacher = nn.parallel.DistributedDataParallel(teacher, device_ids=[args.gpu], find_unused_parameters=True)
        teacher_without_ddp = teacher.module
    else:
        # teacher_without_ddp and teacher are the same thing
        teacher_without_ddp = teacher
    student = nn.parallel.DistributedDataParallel(student, device_ids=[args.gpu], find_unused_parameters=True)
    # # teacher and student start with the same weights
    # teacher_without_ddp.load_state_dict(student.module.state_dict())
    # there is no backpropagation through the teacher, so no need for gradients
    for p in teacher.parameters():
        p.requires_grad = False
    print(f"Student and Teacher are built: they are both {args.arch} network.")


    # ============ preparing loss ... ============
    dino_loss = DINOLoss(
        args.out_dim,
        args.local_crops_number + 2,  # total number of crops = 2 global crops + local_crops_number
        args.warmup_teacher_temp,
        args.teacher_temp,
        args.warmup_teacher_temp_epochs,
        args.epochs,
    ).cuda()
    bce_loss = nn.BCEWithLogitsLoss()
```

```
    # ============ preparing optimizer ... ============
    params_groups = utils.get_params_groups(student)
    if args.optimizer == "adamw":
        optimizer = torch.optim.AdamW(params_groups)  # to use with ViTs
    elif args.optimizer == "sgd":
        optimizer = torch.optim.SGD(params_groups, lr=0, momentum=0.9)  # lr is set by
scheduler
    elif args.optimizer == "lars":
        optimizer = utils.LARS(params_groups)  # to use with convnet and large batches
    # for mixed precision training
    fp16_scaler = None
    if args.use_fp16:
        fp16_scaler = torch.cuda.amp.GradScaler()


    # ============ init schedulers ... ============
    lr_schedule = utils.cosine_scheduler(
        args.lr * (args.batch_size_per_gpu * utils.get_world_size()) / 16.,  # linear scaling rule
        args.min_lr,
        args.epochs, len(data_loader),
        warmup_epochs=args.warmup_epochs,
    )
    wd_schedule = utils.cosine_scheduler(
        args.weight_decay,
        args.weight_decay_end,
        args.epochs, len(data_loader),
    )
```

```
# momentum parameter is increased to 1. during training with a cosine schedule
momentum_schedule = utils.cosine_scheduler(args.momentum_teacher, 1,
                                           args.epochs, len(data_loader))
print(f"Loss, optimizer and schedulers ready for unlabeled batch.")

# ============ optionally resume training ... ============
to_restore = {"epoch": 0}
utils.restart_from_checkpoint(
    os.path.join(args.output_dir, "checkpoint.pth"),
    run_variables=to_restore,
    student=student,
    teacher=teacher,
    optimizer=optimizer,
    fp16_scaler=fp16_scaler,
    dino_loss=dino_loss,
)
start_epoch = to_restore["epoch"]

start_time = time.time()
print("Starting DISTL training !")
for epoch in range(start_epoch, args.epochs):
    data_loader.sampler.set_epoch(epoch)

    # ============ training one epoch of DINO ... ============
    print("============ Training with pseudolabel ... ============")
```

```
    train_stats = train_one_epoch(student, teacher, teacher_without_ddp, dino_loss,
bce_loss,
        data_loader, l_data_loader, optimizer, lr_schedule, wd_schedule,
momentum_schedule,
        epoch, args.ssl_epoch, fp16_scaler, args)


    # ============ writing logs ... ============
    save_dict = {
        'student': student.state_dict(),
        'teacher': teacher.state_dict(),
        'optimizer': optimizer.state_dict(),
        'epoch': epoch + 1,
        'args': args,
        'dino_loss': dino_loss.state_dict(),
    }
    if fp16_scaler is not None:
        save_dict['fp16_scaler'] = fp16_scaler.state_dict()
    utils.save_on_master(save_dict, os.path.join(args.output_dir, 'checkpoint.pth'))
    if args.saveckp_freq and epoch % args.saveckp_freq == 0:
        utils.save_on_master(save_dict, os.path.join(args.output_dir,
f'checkpoint{epoch:04}.pth'))
    log_stats = {**{f'train_{k}': v for k, v in train_stats.items()},
                 'epoch': epoch}
    if utils.is_main_process():
        with (Path(args.output_dir) / "log.txt").open("a") as f:
            f.write(json.dumps(log_stats) + "\n")
  total_time = time.time() - start_time
```

```
    total_time_str = str(datetime.timedelta(seconds=int(total_time)))
    print('Training time {}'.format(total_time_str))


def train_one_epoch(student, teacher, teacher_without_ddp, dino_loss, bce_loss,
data_loader, l_data_loader,
                    optimizer, lr_schedule, wd_schedule, momentum_schedule, epoch, ssl_epoch,
                    fp16_scaler, args):
    metric_logger = utils.MetricLogger(delimiter="  ")
    header = 'Epoch: [{}/{}]'.format(epoch, args.epochs)
    for it, (images, _) in enumerate(metric_logger.log_every(data_loader, 10, header)):
        # update weight decay and learning rate according to their schedule
        it = len(data_loader) * epoch + it  # global training iteration
        for i, param_group in enumerate(optimizer.param_groups):
            param_group["lr"] = lr_schedule[it]
            if i == 0:  # only the first group is regularized
                param_group["weight_decay"] = wd_schedule[it]


        # move images to gpu
        images = [im.cuda(non_blocking=True) for im in images]
        # teacher and student forward passes + compute dino loss
        with torch.cuda.amp.autocast(fp16_scaler is not None):
            teacher_output, teacher_cls = teacher(images[:2])  # only the 2 global views pass
through the teacher
            student_output, student_cls = student(images)
            loss_dino = dino_loss(student_output, teacher_output, epoch)
```

```
        loss_ce = bce_loss(student_cls[args.batch_size_per_gpu:].view(-1),
torch.sigmoid(teacher_cls[:args.batch_size_per_gpu].view(-1)))


        if args.alpha:
            loss_ce = utils.alpha_weight(epoch, args.epochs) * loss_ce
        else:
            loss_ce = loss_ce

        # loss_ce_2 = bce_loss(student_cls[:args.batch_size_per_gpu].view(-1),
torch.sigmoid(teacher_cls[:args.batch_size_per_gpu].view(-1)))
        # loss_ce = (loss_ce_1 + loss_ce_2) / 2.


        if epoch < ssl_epoch:
            loss = args.lam * loss_dino + (1 - args.lam) * loss_ce
        else:
            loss = loss_ce


    if not math.isfinite(loss.item()):
        print("Loss is {}, stopping training".format(loss.item()), force=True)
        sys.exit(1)


    # student update
    optimizer.zero_grad()
    param_norms = None
    if fp16_scaler is None:
        loss.backward()
        if args.clip_grad:
```

```
            param_norms = utils.clip_gradients(student, args.clip_grad)
            utils.cancel_gradients_last_layer(epoch, student,
                                              args.freeze_last_layer)
            optimizer.step()
        else:
            fp16_scaler.scale(loss).backward()
            if args.clip_grad:
                fp16_scaler.unscale_(optimizer)  # unscale the gradients of optimizer's assigned params in-place
                param_norms = utils.clip_gradients(student, args.clip_grad)
            utils.cancel_gradients_last_layer(epoch, student,
                                              args.freeze_last_layer)
            fp16_scaler.step(optimizer)
            fp16_scaler.update()

        # EMA update for the teacher
        with torch.no_grad():
            m = momentum_schedule[it]  # momentum parameter
            for param_q, param_k in zip(student.module.parameters(), teacher_without_ddp.parameters()):
                param_k.data.mul_(m).add_((1 - m) * param_q.detach().data)



        if it % args.correct == 0:
            print("============ Correction with labeled data ... ============")
            # ====================== Correction with labeled data ==============================
```

```
        for _, (images, labels) in enumerate(metric_logger.log_every(l_data_loader, 10, header)):
            # move images to gpu
            images = [im.cuda(non_blocking=True) for im in images]
            labels = labels.float().cuda()

            # teacher and student forward passes + compute dino loss
            with torch.cuda.amp.autocast(fp16_scaler is not None):
                teacher_output, teacher_cls = teacher(images[:2])
                student_output, student_cls = student(images)

                l_loss_dino = dino_loss(student_output, teacher_output, epoch)
                l_loss_ce = bce_loss(student_cls[args.batch_size_per_gpu:].view(-1), labels.view(-1))

                if epoch < ssl_epoch:
                    loss = args.lam * l_loss_dino + (1 - args.lam) * l_loss_ce
                else:
                    loss = l_loss_ce

            if not math.isfinite(loss.item()):
                print("Loss is {}, stopping training".format(loss.item()), force=True)
                sys.exit(1)

            # student update
            optimizer.zero_grad()
            param_norms = None
```

```
            if fp16_scaler is None:
                loss.backward()
                if args.clip_grad:
                    param_norms = utils.clip_gradients(student, args.clip_grad)
                utils.cancel_gradients_last_layer(epoch, student,
                                                  args.freeze_last_layer)
                optimizer.step()
            else:
                fp16_scaler.scale(loss).backward()
                if args.clip_grad:
                    fp16_scaler.unscale_(optimizer)  # unscale the gradients of optimizer's assigned params in-place
                    param_norms = utils.clip_gradients(student, args.clip_grad)
                utils.cancel_gradients_last_layer(epoch, student,
                                                  args.freeze_last_layer)
                fp16_scaler.step(optimizer)
                fp16_scaler.update()

        # logging
        torch.cuda.synchronize()
        metric_logger.update(loss=loss.item())
        metric_logger.update(lr=optimizer.param_groups[0]["lr"])
        metric_logger.update(wd=optimizer.param_groups[0]["weight_decay"])
    # gather the stats from all processes
    metric_logger.synchronize_between_processes()
    print("Averaged stats:", metric_logger)
```

```
    return {k: meter.global_avg for k, meter in metric_logger.meters.items()}



class DINOLoss(nn.Module):
    def __init__(self, out_dim, ncrops, warmup_teacher_temp, teacher_temp,
                 warmup_teacher_temp_epochs, nepochs, student_temp=0.1,
                 center_momentum=0.9):
        super().__init__()
        self.student_temp = student_temp
        self.center_momentum = center_momentum
        self.ncrops = ncrops
        self.register_buffer("center", torch.zeros(1, out_dim))
        # we apply a warm up for the teacher temperature because
        # a too high temperature makes the training instable at the beginning
        self.teacher_temp_schedule = np.concatenate((
            np.linspace(warmup_teacher_temp,
                        teacher_temp, warmup_teacher_temp_epochs),
            np.ones(nepochs - warmup_teacher_temp_epochs) * teacher_temp
        ))

    def forward(self, student_output, teacher_output, epoch):
        """
        Cross-entropy between softmax outputs of the teacher and student networks.
        """
        student_out = student_output / self.student_temp
        student_out = student_out.chunk(self.ncrops)
```

```
    # teacher centering and sharpening
    temp = self.teacher_temp_schedule[epoch]
    teacher_out = F.softmax((teacher_output - self.center) / temp, dim=-1)
    teacher_out = teacher_out.detach().chunk(2)

    total_loss = 0
    n_loss_terms = 0
    for iq, q in enumerate(teacher_out):
        for v in range(len(student_out)):
            if v == iq:
                # we skip cases where student and teacher operate on the same view
                continue
            loss = torch.sum(-q * F.log_softmax(student_out[v], dim=-1), dim=-1)
            total_loss += loss.mean()
            n_loss_terms += 1
    total_loss /= n_loss_terms
    self.update_center(teacher_output)
    return total_loss

@torch.no_grad()
def update_center(self, teacher_output):
    """
    Update center used for teacher output.
    """
    batch_center = torch.sum(teacher_output, dim=0, keepdim=True)
```

```
        dist.all_reduce(batch_center)
        batch_center = batch_center / (len(teacher_output) * dist.get_world_size())

        # ema update
        self.center = self.center * self.center_momentum + batch_center * (1 -
self.center_momentum)


class DataAugmentationDINO(object):
    def __init__(self, global_crops_scale, local_crops_scale, local_crops_number):
        flip_and_color_jitter = transforms.Compose([
            transforms.RandomHorizontalFlip(p=0.5)]
        )

        normalize = transforms.Compose([
            transforms.ToTensor(),
        ])

        # first global crop (Very little augmentation)
        self.global_transfo1 = transforms.Compose([
            normalize,
        ])

        # second global crop
        self.global_transfo2 = transforms.Compose([
            transforms.RandomResizedCrop(256, scale=global_crops_scale,
interpolation=Image.BICUBIC),
```

```
            flip_and_color_jitter,
            transforms.RandomRotation(degrees=(-15, 15)),
            transforms.RandomAutocontrast(p=0.3),
            transforms.RandomEqualize(p=0.3),
            utils.GaussianBlur(0.3),
            normalize,
        ])
        # transformation for the local small crops
        self.local_crops_number = local_crops_number
        self.local_transfo = transforms.Compose([
            transforms.RandomResizedCrop(128, scale=local_crops_scale,
interpolation=Image.BICUBIC),
            flip_and_color_jitter,
            transforms.RandomRotation(degrees=(-15, 15)),
            transforms.RandomAutocontrast(p=0.5),
            transforms.RandomEqualize(p=0.5),
            utils.GaussianBlur(0.5),
            normalize,
        ])


    def __call__(self, image):
        crops = []
        crops.append(self.global_transfo1(image))
        crops.append(self.global_transfo2(image))
        for _ in range(self.local_crops_number):
            crops.append(self.local_transfo(image))
```

```
        return crops



if __name__ == '__main__':
    parser = argparse.ArgumentParser('DINO', parents=[get_args_parser()])
    args = parser.parse_args()
    args.option_dir = './'
    os.makedirs(args.option_dir, exist_ok=True)
    with open(os.path.join(args.option_dir, args.name + '_argv.csv'), 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerows(vars(args).items())
    Path(args.output_dir).mkdir(parents=True, exist_ok=True)


    train_dino(args)


import os
import sys
import datetime
import time
import math
import json
from pathlib import Path
import csv


import numpy as np
from PIL import Image
```

```
import torch
import torch.nn as nn
import torch.distributed as dist
import torch.backends.cudnn as cudnn
import torch.nn.functional as F
from torchvision import transforms
from torchvision import models as torchvision_models

import utils
import vision_transformer as vit_o
from vision_transformer import DINOHead, CLS_head

import CXR_dataset

import main_run

torchvision_archs = sorted(name for name in torchvision_models.__dict__
                    if name.islower() and not name.startswith("__")
                    and callable(torchvision_models.__dict__[name]))

def train_dino(args):
   utils.init_distributed_mode(args)
   utils.fix_random_seeds(args.seed)
   print("git:\n  {}\n".format(utils.get_sha()))
   print("\n".join("%s: %s" % (k, str(v)) for k, v in sorted(dict(vars(args)).items())))
   cudnn.benchmark = True
```

```
    # ============ preparing data ... ============
    transform = DataAugmentationDINO(
        args.global_crops_scale,
        args.local_crops_scale,
        args.local_crops_number,
    )
    dataset = CXR_dataset.CXR_Dataset(args.data_path, transforms=transform,
mode='train', labeled=True)
    sampler = torch.utils.data.DistributedSampler(dataset, shuffle=True)
    data_loader = torch.utils.data.DataLoader(
        dataset,
        sampler=sampler,
        batch_size=args.batch_size_per_gpu,
        num_workers=args.num_workers,
        pin_memory=True,
        drop_last=True,
    )
    print(f"Data loaded: there are {len(dataset)} images.")


    # ============ building student and teacher networks ... ============
    if args.arch in vit_o.__dict__.keys():
        student = vit_o.__dict__[args.arch](
            patch_size=args.patch_size,
            drop_path_rate=args.drop_path_rate,  # stochastic depth
        )
```

```
        teacher = vit_o.__dict__[args.arch](patch_size=args.patch_size)
        embed_dim = student.embed_dim

        state_dict = torch.load(args.pretrained_dir, map_location="cpu")['state_dict']
        # remove `module.` prefix
        state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
        # remove `backbone.` prefix induced by multicrop wrapper
        state_dict = {k.replace("backbone.", ""): v for k, v in state_dict.items()}
        # remove `dino.`
        state_dict = {k.replace("dino.", ""): v for k, v in state_dict.items()}
        msg_t = teacher.load_state_dict(state_dict, strict=False)
        print('(Teacher) Pretrained weights found at {} and loaded with msg:
{}'.format('CheXpert', msg_t))
        msg_s = student.load_state_dict(state_dict, strict=False)
        print('(Student) Pretrained weights found at {} and loaded with msg:
{}'.format('CheXpert', msg_s))

        inter_dim = 384

    # otherwise, we check if the architecture is in torchvision models
    elif args.arch in torchvision_models.__dict__.keys():
        student = torchvision_models.__dict__[args.arch]()
        teacher = torchvision_models.__dict__[args.arch]()

        if 'resne' in args.arch:
            embed_dim = student.fc.weight.shape[1]
            inter_dim = 2048
```

```
    elif 'densenet' in args.arch:
        embed_dim = student.classifier.weight.shape[1]
        inter_dim = 1920

    elif 'eff' in args.arch:
        embed_dim = student.classifier[1].weight.shape[1]
        inter_dim = 1792

    state_dict = torch.load(args.pretrained_dir, map_location="cpu")['state_dict']
    # remove `module.` prefix
    state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
    # remove `backbone.` prefix induced by multicrop wrapper
    state_dict = {k.replace("backbone.", ""): v for k, v in state_dict.items()}
    # remove `dino.`
    state_dict = {k.replace("dino.", ""): v for k, v in state_dict.items()}
    # remove `cnn.`
    state_dict = {k.replace("cnn.", ""): v for k, v in state_dict.items()}
    msg_t = teacher.load_state_dict(state_dict, strict=False)
    print('(Teacher) Pretrained weights found at {} and loaded with msg:
{}'.format('CheXpert', msg_t))
    msg_s = student.load_state_dict(state_dict, strict=False)
    print('(Student) Pretrained weights found at {} and loaded with msg:
{}'.format('CheXpert', msg_s))

  else:
    print(f"Unknow architecture: {args.arch}")
```

```
    # multi-crop wrapper handles forward with inputs of different resolutions
    student = utils.MultiCropWrapper(student, DINOHead(
        embed_dim,
        args.out_dim,
        use_bn=args.use_bn_in_head,
        norm_last_layer=args.norm_last_layer,
    ), CLS_head(inter_dim, 256, 1), args)
    teacher = utils.MultiCropWrapper(
        teacher,
        DINOHead(embed_dim, args.out_dim, args.use_bn_in_head), CLS_head(inter_dim, 256, 1), args
    )

    # move networks to gpu
    student, teacher = student.cuda(), teacher.cuda()
    # synchronize batch norms (if any)
    if utils.has_batchnorms(student):
        student = nn.SyncBatchNorm.convert_sync_batchnorm(student)
        teacher = nn.SyncBatchNorm.convert_sync_batchnorm(teacher)

        # we need DDP wrapper to have synchro batch norms working...
        teacher = nn.parallel.DistributedDataParallel(teacher, device_ids=[args.gpu], find_unused_parameters=True)
        teacher_without_ddp = teacher.module
    else:
        # teacher_without_ddp and teacher are the same thing
```

```
    teacher_without_ddp = teacher
  student = nn.parallel.DistributedDataParallel(student, device_ids=[args.gpu],
find_unused_parameters=True)
  # teacher and student start with the same weights
  teacher_without_ddp.load_state_dict(student.module.state_dict())
  # there is no backpropagation through the teacher, so no need for gradients
  for p in teacher.parameters():
    p.requires_grad = False
  print(f"Student and Teacher are built: they are both {args.arch} network.")

  # ============ preparing loss ... ============
  dino_loss = DINOLoss(
    args.out_dim,
    args.local_crops_number + 2,  # total number of crops = 2 global crops +
local_crops_number
    args.warmup_teacher_temp,
    args.teacher_temp,
    args.warmup_teacher_temp_epochs,
    args.epochs,
  ).cuda()
  bce_loss = nn.BCEWithLogitsLoss()

  # ============ preparing optimizer ... ============
  params_groups = utils.get_params_groups(student)
  if args.optimizer == "adamw":
    optimizer = torch.optim.AdamW(params_groups)  # to use with ViTs
  elif args.optimizer == "sgd":
```

```
    optimizer = torch.optim.SGD(params_groups, lr=0, momentum=0.9)  # lr is set by scheduler
  elif args.optimizer == "lars":
    optimizer = utils.LARS(params_groups)  # to use with convnet and large batches
  # for mixed precision training
  fp16_scaler = None
  if args.use_fp16:
    fp16_scaler = torch.cuda.amp.GradScaler()


  # ============ init schedulers ... ============
  lr_schedule = utils.cosine_scheduler(
    args.lr * (args.batch_size_per_gpu * utils.get_world_size()) / 16.,  # linear scaling rule
    args.min_lr,
    args.epochs, len(data_loader),
    warmup_epochs=args.warmup_epochs,
  )
  wd_schedule = utils.cosine_scheduler(
    args.weight_decay,
    args.weight_decay_end,
    args.epochs, len(data_loader),
  )
  # momentum parameter is increased to 1. during training with a cosine schedule
  momentum_schedule = utils.cosine_scheduler(args.momentum_teacher, 1,
                                             args.epochs, len(data_loader))
  print(f"Loss, optimizer and schedulers ready.")
```

```
    # ============ optionally resume training ... ============
    to_restore = {"epoch": 0}
    utils.restart_from_checkpoint(
        os.path.join(args.output_dir, "checkpoint.pth"),
        run_variables=to_restore,
        student=student,
        teacher=teacher,
        optimizer=optimizer,
        fp16_scaler=fp16_scaler,
        dino_loss=dino_loss,
    )
    start_epoch = to_restore["epoch"]

    start_time = time.time()
    print("Starting pretraining !")
    for epoch in range(start_epoch, args.epochs):
        data_loader.sampler.set_epoch(epoch)

        # ============ training one epoch of DINO ... ============
        train_stats = train_one_epoch(student, teacher, teacher_without_ddp, dino_loss,
bce_loss,
                                      data_loader, optimizer, lr_schedule, wd_schedule,
momentum_schedule,
                                      epoch, args.ssl_epoch, fp16_scaler, args)

        # ============ writing logs ... ============
        save_dict = {
```

```
            'student': student.state_dict(),
            'teacher': teacher.state_dict(),
            'optimizer': optimizer.state_dict(),
            'epoch': epoch + 1,
            'args': args,
            'dino_loss': dino_loss.state_dict(),
        }
        if fp16_scaler is not None:
            save_dict['fp16_scaler'] = fp16_scaler.state_dict()
        utils.save_on_master(save_dict, os.path.join(args.output_dir, 'checkpoint.pth'))
        if args.saveckp_freq and epoch % args.saveckp_freq == 0:
            utils.save_on_master(save_dict, os.path.join(args.output_dir,
f'checkpoint{epoch:04}.pth'))
        log_stats = {**{f'train_{k}': v for k, v in train_stats.items()},
                     'epoch': epoch}
        if utils.is_main_process():
            with (Path(args.output_dir) / "log.txt").open("a") as f:
                f.write(json.dumps(log_stats) + "\n")
    total_time = time.time() - start_time
    total_time_str = str(datetime.timedelta(seconds=int(total_time)))
    print('Training time {}'.format(total_time_str))



def train_one_epoch(student, teacher, teacher_without_ddp, dino_loss, bce_loss,
data_loader,
                    optimizer, lr_schedule, wd_schedule, momentum_schedule, epoch, ssl_epoch,
                    fp16_scaler, args):
```

```
metric_logger = utils.MetricLogger(delimiter="  ")
header = 'Epoch: [{}/{}]'.format(epoch, args.epochs)
for it, (images, labels) in enumerate(metric_logger.log_every(data_loader, 10, header)):
    # update weight decay and learning rate according to their schedule
    it = len(data_loader) * epoch + it  # global training iteration
    for i, param_group in enumerate(optimizer.param_groups):
        param_group["lr"] = lr_schedule[it]
        if i == 0:  # only the first group is regularized
            param_group["weight_decay"] = wd_schedule[it]

    # move images to gpu
    images = [im.cuda(non_blocking=True) for im in images]
    labels = labels.float().cuda()
    # teacher and student forward passes + compute dino loss
    with torch.cuda.amp.autocast(fp16_scaler is not None):
        student_cls = student(images[1])   # Augmented global view image
        loss = bce_loss(student_cls.view(-1), labels.view(-1))

    if not math.isfinite(loss.item()):
        print("Loss is {}, stopping training".format(loss.item()), force=True)
        sys.exit(1)

    # student update
    optimizer.zero_grad()
    param_norms = None
    if fp16_scaler is None:
```

```
            loss.backward()
            if args.clip_grad:
                param_norms = utils.clip_gradients(student, args.clip_grad)
            utils.cancel_gradients_last_layer(epoch, student,
                                              args.freeze_last_layer)
            optimizer.step()
        else:
            fp16_scaler.scale(loss).backward()
            if args.clip_grad:
                fp16_scaler.unscale_(optimizer)  # unscale the gradients of optimizer's assigned params in-place
                param_norms = utils.clip_gradients(student, args.clip_grad)
            utils.cancel_gradients_last_layer(epoch, student,
                                              args.freeze_last_layer)
            fp16_scaler.step(optimizer)
            fp16_scaler.update()

        # EMA update for the teacher
        with torch.no_grad():
            m = momentum_schedule[it]  # momentum parameter
            for param_q, param_k in zip(student.module.parameters(), teacher_without_ddp.parameters()):
                param_k.data.mul_(m).add_((1 - m) * param_q.detach().data)

        # logging
        torch.cuda.synchronize()
        metric_logger.update(loss=loss.item())
```

```
        metric_logger.update(lr=optimizer.param_groups[0]["lr"])
        metric_logger.update(wd=optimizer.param_groups[0]["weight_decay"])
    # gather the stats from all processes
    metric_logger.synchronize_between_processes()
    print("Averaged stats:", metric_logger)
    return {k: meter.global_avg for k, meter in metric_logger.meters.items()}


class DINOLoss(nn.Module):
    def __init__(self, out_dim, ncrops, warmup_teacher_temp, teacher_temp,
                 warmup_teacher_temp_epochs, nepochs, student_temp=0.1,
                 center_momentum=0.9):
        super().__init__()
        self.student_temp = student_temp
        self.center_momentum = center_momentum
        self.ncrops = ncrops
        self.register_buffer("center", torch.zeros(1, out_dim))

        self.teacher_temp_schedule = np.concatenate((
            np.linspace(warmup_teacher_temp,
                        teacher_temp, warmup_teacher_temp_epochs),
            np.ones(nepochs - warmup_teacher_temp_epochs) * teacher_temp
        ))

    def forward(self, student_output, teacher_output, epoch):
        """
```

```
    Cross-entropy between softmax outputs of the teacher and student networks.
    """
    student_out = student_output / self.student_temp
    student_out = student_out.chunk(self.ncrops)

    # teacher centering and sharpening
    temp = self.teacher_temp_schedule[epoch]
    teacher_out = F.softmax((teacher_output - self.center) / temp, dim=-1)
    teacher_out = teacher_out.detach().chunk(2)

    total_loss = 0
    n_loss_terms = 0
    for iq, q in enumerate(teacher_out):
        for v in range(len(student_out)):
            if v == iq:
                # we skip cases where student and teacher operate on the same view
                continue
            loss = torch.sum(-q * F.log_softmax(student_out[v], dim=-1), dim=-1)
            total_loss += loss.mean()
            n_loss_terms += 1
    total_loss /= n_loss_terms
    self.update_center(teacher_output)
    return total_loss

@torch.no_grad()
def update_center(self, teacher_output):
```

```
        """
        Update center used for teacher output.
        """
        batch_center = torch.sum(teacher_output, dim=0, keepdim=True)
        dist.all_reduce(batch_center)
        batch_center = batch_center / (len(teacher_output) * dist.get_world_size())

        # ema update
        self.center = self.center * self.center_momentum + batch_center * (1 -
self.center_momentum)


class DataAugmentationDINO(object):
    def __init__(self, global_crops_scale, local_crops_scale, local_crops_number):
        flip_and_color_jitter = transforms.Compose([
            transforms.RandomHorizontalFlip(p=0.5)]
        )

        normalize = transforms.Compose([
            transforms.ToTensor(),
        ])

        # first global crop (original image)
        self.global_transfo1 = transforms.Compose([
            normalize,
        ])
```

```
        # second global crop (little augmentation)
        self.global_transfo2 = transforms.Compose([
            transforms.RandomResizedCrop(256, scale=global_crops_scale,
interpolation=Image.BICUBIC),
            flip_and_color_jitter,
            transforms.RandomRotation(degrees=(-15, 15)),
            transforms.RandomAutocontrast(p=0.3),
            transforms.RandomEqualize(p=0.3),
            utils.GaussianBlur(0.3),
            normalize,
        ])

        # transformation for the local small crops (multiple augmentations with cropping)
        self.local_crops_number = local_crops_number
        self.local_transfo = transforms.Compose([
            transforms.RandomResizedCrop(128, scale=local_crops_scale,
interpolation=Image.BICUBIC),
            flip_and_color_jitter,
            transforms.RandomRotation(degrees=(-15, 15)),
            transforms.RandomAutocontrast(p=0.5),
            transforms.RandomEqualize(p=0.5),
            utils.GaussianBlur(0.5),
            normalize,
        ])

    def __call__(self, image):
```

```
        crops = []
        crops.append(self.global_transfo1(image))
        crops.append(self.global_transfo2(image))
        for _ in range(self.local_crops_number):
            crops.append(self.local_transfo(image))
        return crops


if __name__ == '__main__':
    parser = main_run.get_args_parser()
    args = parser.parse_args()
    args.option_dir = './'
    os.makedirs(args.option_dir, exist_ok=True)
    with open(os.path.join(args.option_dir, args.name + '_argv.csv'), 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerows(vars(args).items())
    Path(args.output_dir).mkdir(parents=True, exist_ok=True)

    train_dino(args)
```

requirements.txt

```
absl-py==1.0.0
apturl==0.5.2
asn1crypto==0.24.0
```

blessings==1.7

Brlapi==0.6.6

cachetools==5.0.0

certifi==2021.10.8

chardet==3.0.4

charset-normalizer==2.0.11

click==8.1.2

command-not-found==0.3

contextlib2==21.6.0

cryptography==2.1.4

cupshelpers==1.0

cycler==0.11.0

Cython==0.29.28

defer==1.0.6

distro-info===0.18ubuntu0.18.04.1

easydict==1.9

filelock==3.6.0

fonttools==4.29.1

google-auth==2.6.0

google-auth-oauthlib==0.4.6

gpustat==0.6.0

grpcio==1.43.0

httplib2==0.9.2

huggingface-hub==0.0.12

idna==3.3

imageio==2.15.0

importlib-metadata==4.10.1

joblib==1.1.0

keyring==10.6.0

keyrings.alt==3.0

kiwisolver==1.3.2

language-selector==0.1

launchpadlib==1.10.6

lazr.restfulclient==0.13.5

lazr.uri==1.0.3

louis==3.5.0

macaroonbakery==1.1.3

Mako==1.0.7

Markdown==3.3.6

MarkupSafe==1.0

matplotlib==3.5.1

ml-collections==0.1.1

netifaces==0.10.4

networkx==2.6.3

nibabel==3.2.2

numpy==1.21.6

nvidia-ml-py3==7.352.0

oauth==1.0.1

oauthlib==3.2.0

olefile==0.45.1

opencv-python==4.5.5.62

packaging==21.3

pandas==1.0.3

pexpect==4.2.1

Pillow==9.0.1

protobuf==3.19.4

psutil==5.9.0

pyasn1==0.4.8

pyasn1-modules==0.2.8

pycairo==1.16.2

pycrypto==2.6.1

pycups==1.9.73

pydicom==2.3.0

PyGObject==3.26.1

pymacaroons==0.13.0

PyNaCl==1.1.2

pyparsing==3.0.7

pyRFC3339==1.0

python-apt==1.6.5+ubuntu0.5

python-dateutil==2.8.2

python-debian==0.1.32

pytorch-gradcam==0.2.1

pytz==2019.3

PyWavelets==1.2.0

pyxdg==0.25

PyYAML==6.0

regex==2022.4.24

reportlab==3.4.0

requests==2.27.1

requests-oauthlib==1.3.1

requests-unixsocket==0.1.5

rsa==4.8

ruamel.yaml==0.17.21

ruamel.yaml.clib==0.2.6

sacremoses==0.0.49

scikit-image==0.19.1

scikit-learn==1.0.2

scipy==1.7.3

screen-resolution-extra==0.0.0

SecretStorage==2.3.1

SimpleITK==2.1.1

simplejson==3.13.2

six==1.16.0

sklearn==0.0

ssh-import-id==5.7

system-service==0.3

systemd-python==234

tensorboard==2.8.0

tensorboard-data-server==0.6.1

tensorboard-plugin-wit==1.8.1

threadpoolctl==3.1.0

tifffile==2021.11.2

timm==0.5.4

tokenizers==0.10.3

torch==1.8.0+cu111

torchaudio==0.8.0

torchsummary==1.5.1

torchvision==0.9.0+cu111

tqdm==4.62.3

transformers==4.8.1

typing_extensions==4.0.1

ubuntu-drivers-common==0.0.0

ufw==0.36

unattended-upgrades==0.1

urllib3==1.26.8

usb-creator==0.3.3

wadllib==1.3.2

Werkzeug==2.0.3

xkit==0.0.0

zipp==3.7.0

zope.interface==4.3.2

utils.py

```
import os
import sys
import time
import math
import random
import datetime
```

```
import subprocess
from collections import defaultdict, deque

import numpy as np
import torch
from torch import nn
import torch.distributed as dist
from PIL import ImageFilter, ImageOps



class GaussianBlur(object):
  """
  Apply Gaussian Blur to the PIL save.
  """
  def __init__(self, p=0.5, radius_min=0.1, radius_max=2.):
    self.prob = p
    self.radius_min = radius_min
    self.radius_max = radius_max

  def __call__(self, img):
    do_it = random.random() <= self.prob
    if not do_it:
      return img

    return img.filter(
      ImageFilter.GaussianBlur(
```

```
            radius=random.uniform(self.radius_min, self.radius_max)
        )
      )


class GaussianBlurInference(object):
  """
  Apply Gaussian Blur to the PIL save.
  """
  def __init__(self):
    self.radius = 0.5

  def __call__(self, img):

    return img.filter(
      ImageFilter.GaussianBlur(self.radius)
      )


class Solarization(object):
  """
  Apply Solarization to the PIL save.
  """
  def __init__(self, p):
    self.p = p
```

```
  def __call__(self, img):
    if random.random() < self.p:
      return ImageOps.solarize(img)
    else:
      return img


def load_pretrained_weights(model, pretrained_weights, checkpoint_key, model_name, patch_size):
  if os.path.isfile(pretrained_weights):
    state_dict = torch.load(pretrained_weights, map_location="cpu")
    if checkpoint_key is not None and checkpoint_key in state_dict:
      print(f"Take key {checkpoint_key} in provided checkpoint dict")
      state_dict = state_dict[checkpoint_key]
    # remove `module.` prefix
    state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
    # remove `backbone.` prefix induced by multicrop wrapper
    state_dict = {k.replace("backbone.", ""): v for k, v in state_dict.items()}
    msg = model.load_state_dict(state_dict, strict=False)
    print('Pretrained weights found at {} and loaded with msg: {}'.format(pretrained_weights, msg))
  else:
    print("Please use the `--pretrained_weights` argument to indicate the path of the checkpoint to evaluate.")
    url = None
    if model_name == "vit_small" and patch_size == 16:
      url = "dino_deitsmall16_pretrain/dino_deitsmall16_pretrain.pth"
```

```
    elif model_name == "vit_small" and patch_size == 8:
        url = "dino_deitsmall8_pretrain/dino_deitsmall8_pretrain.pth"
    elif model_name == "vit_base" and patch_size == 16:
        url = "dino_vitbase16_pretrain/dino_vitbase16_pretrain.pth"
    elif model_name == "vit_base" and patch_size == 8:
        url = "dino_vitbase8_pretrain/dino_vitbase8_pretrain.pth"
    elif model_name == "xcit_small_12_p16":
        url = "dino_xcit_small_12_p16_pretrain/dino_xcit_small_12_p16_pretrain.pth"
    elif model_name == "xcit_small_12_p8":
        url = "dino_xcit_small_12_p8_pretrain/dino_xcit_small_12_p8_pretrain.pth"
    elif model_name == "xcit_medium_24_p16":
        url = "dino_xcit_medium_24_p16_pretrain/dino_xcit_medium_24_p16_pretrain.pth"
    elif model_name == "xcit_medium_24_p8":
        url = "dino_xcit_medium_24_p8_pretrain/dino_xcit_medium_24_p8_pretrain.pth"
    elif model_name == "resnet50":
        url = "dino_resnet50_pretrain/dino_resnet50_pretrain.pth"
    if url is not None:
        print("Since no pretrained weights have been provided, we load the reference pretrained DINO weights.")
        state_dict = torch.hub.load_state_dict_from_url(url="https://dl.fbaipublicfiles.com/dino/" + url)
        model.load_state_dict(state_dict, strict=True)
    else:
        print("There is no reference weights available for this model => We use random weights.")
```

```
def clip_gradients(model, clip):
    norms = []
    for name, p in model.named_parameters():
        if p.grad is not None:
            param_norm = p.grad.data.norm(2)
            norms.append(param_norm.item())
            clip_coef = clip / (param_norm + 1e-6)
            if clip_coef < 1:
                p.grad.data.mul_(clip_coef)
    return norms


def cancel_gradients_last_layer(epoch, model, freeze_last_layer):
    if epoch >= freeze_last_layer:
        return
    for n, p in model.named_parameters():
        if "last_layer" in n:
            p.grad = None


def restart_from_checkpoint(ckp_path, run_variables=None, **kwargs):
    """
    Re-start from checkpoint
    """
    if not os.path.isfile(ckp_path):
        return
```

```
print("Found checkpoint at {}".format(ckp_path))


# open checkpoint file
checkpoint = torch.load(ckp_path, map_location="cpu")


# key is what to look for in the checkpoint file
# value is the object to load
# example: {'state_dict': model}
for key, value in kwargs.items():
  if key in checkpoint and value is not None:
    try:
      msg = value.load_state_dict(checkpoint[key], strict=False)
      print("=> loaded '{}' from checkpoint '{}' with msg {}".format(key, ckp_path, msg))
    except TypeError:
      try:
        msg = value.load_state_dict(checkpoint[key])
        print("=> loaded '{}' from checkpoint: '{}'".format(key, ckp_path))
      except ValueError:
        print("=> failed to load '{}' from checkpoint: '{}'".format(key, ckp_path))
  else:
    print("=> key '{}' not found in checkpoint: '{}'".format(key, ckp_path))


# re load variable important for the run
if run_variables is not None:
  for var_name in run_variables:
    if var_name in checkpoint:
```

```
            run_variables[var_name] = checkpoint[var_name]


def cosine_scheduler(base_value, final_value, epochs, niter_per_ep, warmup_epochs=0,
start_warmup_value=0):
    warmup_schedule = np.array([])
    warmup_iters = warmup_epochs * niter_per_ep
    if warmup_epochs > 0:
        warmup_schedule = np.linspace(start_warmup_value, base_value, warmup_iters)

    iters = np.arange(epochs * niter_per_ep - warmup_iters)
    schedule = final_value + 0.5 * (base_value - final_value) * (1 + np.cos(np.pi * iters /
len(iters)))

    schedule = np.concatenate((warmup_schedule, schedule))
    assert len(schedule) == epochs * niter_per_ep
    return schedule


def bool_flag(s):
    """
    Parse boolean arguments from the command line.
    """
    FALSY_STRINGS = {"off", "false", "0"}
    TRUTHY_STRINGS = {"on", "true", "1"}
    if s.lower() in FALSY_STRINGS:
        return False
```

```
    elif s.lower() in TRUTHY_STRINGS:
        return True
    else:
        raise argparse.ArgumentTypeError("invalid value for a boolean flag")


def fix_random_seeds(seed=31):
    """
    Fix random seeds.
    """
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)


class SmoothedValue(object):
    """Track a series of values and provide access to smoothed values over a
    window or the global series average.
    """

    def __init__(self, window_size=20, fmt=None):
        if fmt is None:
            fmt = "{median:.6f} ({global_avg:.6f})"
        self.deque = deque(maxlen=window_size)
        self.total = 0.0
        self.count = 0
```

```
    self.fmt = fmt

def update(self, value, n=1):
    self.deque.append(value)
    self.count += n
    self.total += value * n

def synchronize_between_processes(self):
    """
    Warning: does not synchronize the deque!
    """
    if not is_dist_avail_and_initialized():
        return
    t = torch.tensor([self.count, self.total], dtype=torch.float64, device='cuda')
    dist.barrier()
    dist.all_reduce(t)
    t = t.tolist()
    self.count = int(t[0])
    self.total = t[1]

@property
def median(self):
    d = torch.tensor(list(self.deque))
    return d.median().item()

@property
```

```
    def avg(self):
        d = torch.tensor(list(self.deque), dtype=torch.float32)
        return d.mean().item()

    @property
    def global_avg(self):
        return self.total / self.count

    @property
    def max(self):
        return max(self.deque)

    @property
    def value(self):
        return self.deque[-1]

    def __str__(self):
        return self.fmt.format(
            median=self.median,
            avg=self.avg,
            global_avg=self.global_avg,
            max=self.max,
            value=self.value)


def reduce_dict(input_dict, average=True):
```

```
"""
Args:
    input_dict (dict): all the values will be reduced
    average (bool): whether to do average or sum
Reduce the values in the dictionary from all processes so that all processes
have the averaged results. Returns a dict with the same fields as
input_dict, after reduction.
"""
world_size = get_world_size()
if world_size < 2:
    return input_dict
with torch.no_grad():
    names = []
    values = []
    # sort the keys so that they are consistent across processes
    for k in sorted(input_dict.keys()):
        names.append(k)
        values.append(input_dict[k])
    values = torch.stack(values, dim=0)
    dist.all_reduce(values)
    if average:
        values /= world_size
    reduced_dict = {k: v for k, v in zip(names, values)}
return reduced_dict
```

```
class MetricLogger(object):
  def __init__(self, delimiter="\t"):
    self.meters = defaultdict(SmoothedValue)
    self.delimiter = delimiter

  def update(self, **kwargs):
    for k, v in kwargs.items():
      if isinstance(v, torch.Tensor):
        v = v.item()
      assert isinstance(v, (float, int))
      self.meters[k].update(v)

  def __getattr__(self, attr):
    if attr in self.meters:
      return self.meters[attr]
    if attr in self.__dict__:
      return self.__dict__[attr]
    raise AttributeError("'{}' object has no attribute '{}'".format(
      type(self).__name__, attr))

  def __str__(self):
    loss_str = []
    for name, meter in self.meters.items():
      loss_str.append(
        "{}: {}".format(name, str(meter))
      )
```

```
    return self.delimiter.join(loss_str)

def synchronize_between_processes(self):
    for meter in self.meters.values():
        meter.synchronize_between_processes()

def add_meter(self, name, meter):
    self.meters[name] = meter

def log_every(self, iterable, print_freq, header=None):
    i = 0
    if not header:
        header = ''
    start_time = time.time()
    end = time.time()
    iter_time = SmoothedValue(fmt='{avg:.6f}')
    data_time = SmoothedValue(fmt='{avg:.6f}')
    space_fmt = ':' + str(len(str(len(iterable)))) + 'd'
    if torch.cuda.is_available():
        log_msg = self.delimiter.join([
            header,
            '[{0' + space_fmt + '}/{1}]',
            'eta: {eta}',
            '{meters}',
            'time: {time}',
            'data: {data}',
```

```
        'max mem: {memory:.0f}'
    ])
else:
    log_msg = self.delimiter.join([
        header,
        '[{0' + space_fmt + '}/{1}]',
        'eta: {eta}',
        '{meters}',
        'time: {time}',
        'data: {data}'
    ])
MB = 1024.0 * 1024.0
for obj in iterable:
    data_time.update(time.time() - end)
    yield obj
    iter_time.update(time.time() - end)
    if i % print_freq == 0 or i == len(iterable) - 1:
        eta_seconds = iter_time.global_avg * (len(iterable) - i)
        eta_string = str(datetime.timedelta(seconds=int(eta_seconds)))
        if torch.cuda.is_available():
            print(log_msg.format(
                i, len(iterable), eta=eta_string,
                meters=str(self),
                time=str(iter_time), data=str(data_time),
                memory=torch.cuda.max_memory_allocated() / MB))
        else:
```

```
            print(log_msg.format(
              i, len(iterable), eta=eta_string,
              meters=str(self),
              time=str(iter_time), data=str(data_time)))
        i += 1
        end = time.time()
    total_time = time.time() - start_time
    total_time_str = str(datetime.timedelta(seconds=int(total_time)))
    print('{} Total time: {} ({:.6f} s / it)'.format(
      header, total_time_str, total_time / len(iterable)))



def get_sha():
  cwd = os.path.dirname(os.path.abspath(__file__))

  def _run(command):
    return subprocess.check_output(command, cwd=cwd).decode('ascii').strip()
  sha = 'N/A'
  diff = "clean"
  branch = 'N/A'
  try:
    sha = _run(['git', 'rev-parse', 'HEAD'])
    subprocess.check_output(['git', 'diff'], cwd=cwd)
    diff = _run(['git', 'diff-index', 'HEAD'])
    diff = "has uncommited changes" if diff else "clean"
    branch = _run(['git', 'rev-parse', '--abbrev-ref', 'HEAD'])
```

```
    except Exception:
        pass
    message = f"sha: {sha}, status: {diff}, branch: {branch}"
    return message


def is_dist_avail_and_initialized():
    if not dist.is_available():
        return False
    if not dist.is_initialized():
        return False
    return True


def get_world_size():
    if not is_dist_avail_and_initialized():
        return 1
    return dist.get_world_size()


def get_rank():
    if not is_dist_avail_and_initialized():
        return 0
    return dist.get_rank()
```

```
def is_main_process():
    return get_rank() == 0


def save_on_master(*args, **kwargs):
    if is_main_process():
        torch.save(*args, **kwargs)


def setup_for_distributed(is_master):
    """
    This function disables printing when not in master process
    """
    import builtins as __builtin__
    builtin_print = __builtin__.print

    def print(*args, **kwargs):
        force = kwargs.pop('force', False)
        if is_master or force:
            builtin_print(*args, **kwargs)

    __builtin__.print = print


def init_distributed_mode(args):
    # launched with torch.distributed.launch
```

```
if 'RANK' in os.environ and 'WORLD_SIZE' in os.environ:
    args.rank = int(os.environ["RANK"])
    args.world_size = int(os.environ['WORLD_SIZE'])
    args.gpu = int(os.environ['LOCAL_RANK'])
# launched with submitit on a slurm cluster
elif 'SLURM_PROCID' in os.environ:
    args.rank = int(os.environ['SLURM_PROCID'])
    args.gpu = args.rank % torch.cuda.device_count()
# launched naively with `python main_run.py`
# we manually add MASTER_ADDR and MASTER_PORT to env variables
elif torch.cuda.is_available():
    print('Will run the code on one GPU.')
    args.rank, args.gpu, args.world_size = 0, 0, 1
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
else:
    print('Does not support training without GPU.')
    sys.exit(1)

dist.init_process_group(
    backend="nccl",
    init_method=args.dist_url,
    world_size=args.world_size,
    rank=args.rank,
)
```

```
    torch.cuda.set_device(args.gpu)
    print('| distributed init (rank {}): {}'.format(
        args.rank, args.dist_url), flush=True)
    dist.barrier()
    setup_for_distributed(args.rank == 0)


def accuracy(output, target, topk=(1,)):
    """Computes the accuracy over the k top predictions for the specified values of k"""
    maxk = max(topk)
    batch_size = target.size(0)
    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.reshape(1, -1).expand_as(pred))
    return [correct[:k].reshape(-1).float().sum(0) * 100. / batch_size for k in topk]


def _no_grad_trunc_normal_(tensor, mean, std, a, b):
    # Cut & paste from PyTorch official master until it's in a few official releases - RW
    # Method based on
https://people.sc.fsu.edu/~jburkardt/presentations/truncated_normal.pdf
    def norm_cdf(x):
        # Computes standard normal cumulative distribution function
        return (1. + math.erf(x / math.sqrt(2.))) / 2.

    if (mean < a - 2 * std) or (mean > b + 2 * std):
```

```
    warnings.warn("mean is more than 2 std from [a, b] in nn.init.trunc_normal_. "
                  "The distribution of values may be incorrect.",
                  stacklevel=2)

with torch.no_grad():
    # Values are generated by using a truncated uniform distribution and
    # then using the inverse CDF for the normal distribution.
    # Get upper and lower cdf values
    l = norm_cdf((a - mean) / std)
    u = norm_cdf((b - mean) / std)

    # Uniformly fill tensor with values from [l, u], then translate to
    # [2l-1, 2u-1].
    tensor.uniform_(2 * l - 1, 2 * u - 1)

    # Use inverse cdf transform for normal distribution to get truncated
    # standard normal
    tensor.erfinv_()

    # Transform to proper mean, std
    tensor.mul_(std * math.sqrt(2.))
    tensor.add_(mean)

    # Clamp to ensure it's in the proper range
    tensor.clamp_(min=a, max=b)
    return tensor
```

```
def trunc_normal_(tensor, mean=0., std=1., a=-2., b=2.):
    # type: (Tensor, float, float, float, float) -> Tensor
    return _no_grad_trunc_normal_(tensor, mean, std, a, b)



class LARS(torch.optim.Optimizer):
    """
    Almost copy-paste from
https://github.com/facebookresearch/barlowtwins/blob/main/main.py
    """
    def __init__(self, params, lr=0, weight_decay=0, momentum=0.9, eta=0.001,
                 weight_decay_filter=None, lars_adaptation_filter=None):
        defaults = dict(lr=lr, weight_decay=weight_decay, momentum=momentum,
                        eta=eta, weight_decay_filter=weight_decay_filter,
                        lars_adaptation_filter=lars_adaptation_filter)
        super().__init__(params, defaults)

    @torch.no_grad()
    def step(self):
        for g in self.param_groups:
            for p in g['params']:
                dp = p.grad

                if dp is None:
```

```
                continue

            if p.ndim != 1:
                dp = dp.add(p, alpha=g['weight_decay'])

            if p.ndim != 1:
                param_norm = torch.norm(p)
                update_norm = torch.norm(dp)
                one = torch.ones_like(param_norm)
                q = torch.where(param_norm > 0.,
                            torch.where(update_norm > 0,
                                    (g['eta'] * param_norm / update_norm), one), one)
                dp = dp.mul(q)

            param_state = self.state[p]
            if 'mu' not in param_state:
                param_state['mu'] = torch.zeros_like(p)
            mu = param_state['mu']
            mu.mul_(g['momentum']).add_(dp)

            p.add_(mu, alpha=-g['lr'])



class MultiCropWrapper(nn.Module):
    """
    Perform forward pass separately on each resolution input.
```

The inputs corresponding to a single resolution are clubbed and single forward is run on the same resolution inputs. Hence we do several forward passes = number of different resolutions used. We then concatenate all the output features and run the head forward on these concatenated features.

"""

```
def __init__(self, backbone, head, cls_head, args):
    super(MultiCropWrapper, self).__init__()
    # disable layers dedicated to ImageNet labels classification
    # backbone.fc, backbone.head = nn.Identity(), nn.Identity()
    # self.backbone = backbone
    self.head = head
    self.cls_head = cls_head
    self.args = args

    if 'densenet' in self.args.arch:
        backbone.classifier = nn.Identity()
        self.backbone = backbone
    elif 'resne' in self.args.arch:
        backbone.fc = nn.Identity()
        self.backbone = backbone
    elif 'eff' in self.args.arch:
        backbone.classifier = nn.Identity()
        self.backbone = backbone
    else:
        backbone.fc, backbone.head = nn.Identity(), nn.Identity()
```

```
        self.backbone = backbone


    def forward(self, x):
        # convert to list
        if not isinstance(x, list):
            x = [x]
        idx_crops = torch.cumsum(torch.unique_consecutive(
            torch.tensor([inp.shape[-1] for inp in x]),
            return_counts=True,
        )[1], 0)

        # # For Deit
        # if len(x) <= 1:
        #     pass
        # else:
        #     idx_crops = [1, 4]

        start_idx, output = 0, torch.empty(0).to(x[0].device)
        for end_idx in idx_crops:
            if 'deit' in self.args.arch:
                _out = self.backbone.forward_features(torch.cat(x[start_idx: end_idx]))
            else:
                _out = self.backbone(torch.cat(x[start_idx: end_idx]))
            # The output is a tuple with XCiT model. See:
            # https://github.com/facebookresearch/xcit/blob/master/xcit.py#L404-L405
            if isinstance(_out, tuple):
```

```
            _out = _out[0]
        # accumulate outputs
        output = torch.cat((output, _out))
        start_idx = end_idx

        # only first two global views for student
        if end_idx <= 2:
            cls_output = self.cls_head(output)

    # Run the head forward on the concatenated features.
    if end_idx == 1:
        return cls_output
    else:
        return self.head(output), cls_output


def get_params_groups(model):
    regularized = []
    not_regularized = []
    for name, param in model.named_parameters():
        if not param.requires_grad:
            continue
        # we do not regularize biases nor Norm parameters
        if name.endswith(".bias") or len(param.shape) == 1:
            not_regularized.append(param)
        else:
```

```
            regularized.append(param)
    return [{'params': regularized}, {'params': not_regularized, 'weight_decay': 0.}]



def has_batchnorms(model):
    bn_types = (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d, nn.SyncBatchNorm)
    for name, module in model.named_modules():
        if isinstance(module, bn_types):
            return True
    return False



class PCA():
    """
    Class to  compute and apply PCA.
    """
    def __init__(self, dim=256, whit=0.5):
        self.dim = dim
        self.whit = whit
        self.mean = None

    def train_pca(self, cov):
        """
        Takes a covariance matrix (np.ndarray) as input.
        """
        d, v = np.linalg.eigh(cov)
```

```
    eps = d.max() * 1e-5
    n_0 = (d < eps).sum()
    if n_0 > 0:
        d[d < eps] = eps

    # total energy
    totenergy = d.sum()

    # sort eigenvectors with eigenvalues order
    idx = np.argsort(d)[::-1][:self.dim]
    d = d[idx]
    v = v[:, idx]

    print("keeping %.2f %% of the energy" % (d.sum() / totenergy * 100.0))

    # for the whitening
    d = np.diag(1. / d**self.whit)

    # principal components
    self.dvt = np.dot(d, v.T)

def apply(self, x):
    # input is from numpy
    if isinstance(x, np.ndarray):
        if self.mean is not None:
            x -= self.mean
```

```
        return np.dot(self.dvt, x.T).T

    # input is from torch and is on GPU
    if x.is_cuda:
        if self.mean is not None:
            x -= torch.cuda.FloatTensor(self.mean)
        return torch.mm(torch.cuda.FloatTensor(self.dvt), x.transpose(0, 1)).transpose(0, 1)

    # input if from torch, on CPU
    if self.mean is not None:
        x -= torch.FloatTensor(self.mean)
    return torch.mm(torch.FloatTensor(self.dvt), x.transpose(0, 1)).transpose(0, 1)


def compute_ap(ranks, nres):
    """
    Computes average precision for given ranked indexes.
    Arguments
    ---------
    ranks : zerro-based ranks of positive images
    nres  : number of positive images
    Returns
    -------
    ap    : average precision
    """
```

```
    # number of images ranked by the system
    nimgranks = len(ranks)

    # accumulate trapezoids in PR-plot
    ap = 0

    recall_step = 1. / nres

    for j in np.arange(nimgranks):
        rank = ranks[j]

        if rank == 0:
            precision_0 = 1.
        else:
            precision_0 = float(j) / rank

        precision_1 = float(j + 1) / (rank + 1)

        ap += (precision_0 + precision_1) * recall_step / 2.

    return ap


def compute_map(ranks, gnd, kappas=[]):
    """
    Computes the mAP for a given set of returned results.
```

```
    Usage:
      map = compute_map (ranks, gnd)
           computes mean average precsion (map) only
      map, aps, pr, prs = compute_map (ranks, gnd, kappas)
           computes mean average precision (map), average precision (aps) for each query
           computes mean precision at kappas (pr), precision at kappas (prs) for each query
    Notes:
    1) ranks starts from 0, ranks.shape = db_size X #queries
    2) The junk results (e.g., the query itself) should be declared in the gnd stuct array
    3) If there are no positive images for some query, that query is excluded from the evaluation
    """

    map = 0.
    nq = len(gnd) # number of queries
    aps = np.zeros(nq)
    pr = np.zeros(len(kappas))
    prs = np.zeros((nq, len(kappas)))
    nempty = 0

    for i in np.arange(nq):
        qgnd = np.array(gnd[i]['ok'])

        # no positive images, skip from the average
        if qgnd.shape[0] == 0:
            aps[i] = float('nan')
```

```
    prs[i, :] = float('nan')
    nempty += 1
    continue

try:
    qgndj = np.array(gnd[i]['junk'])
except:
    qgndj = np.empty(0)

# sorted positions of positive and junk images (0 based)
pos  = np.arange(ranks.shape[0])[np.in1d(ranks[:,i], qgnd)]
junk = np.arange(ranks.shape[0])[np.in1d(ranks[:,i], qgndj)]

k = 0;
ij = 0;
if len(junk):
    # decrease positions of positives based on the number of
    # junk images appearing before them
    ip = 0
    while (ip < len(pos)):
        while (ij < len(junk) and pos[ip] > junk[ij]):
            k += 1
            ij += 1
        pos[ip] = pos[ip] - k
        ip += 1
```

```
        # compute ap
        ap = compute_ap(pos, len(qgnd))
        map = map + ap
        aps[i] = ap

        # compute precision @ k
        pos += 1 # get it to 1-based
        for j in np.arange(len(kappas)):
            kq = min(max(pos), kappas[j]);
            prs[i, j] = (pos <= kq).sum() / kq
        pr = pr + prs[i, :]

    map = map / (nq - nempty)
    pr = pr / (nq - nempty)

    return map, aps, pr, prs



def multi_scale(samples, model):
    v = None
    for s in [1, 1/2**(1/2), 1/2]:  # we use 3 different scales
        if s == 1:
            inp = samples.clone()
        else:
            inp = nn.functional.interpolate(samples, scale_factor=s, mode='bilinear',
align_corners=False)
```

```
            feats = model(inp).clone()
            if v is None:
                v = feats
            else:
                v += feats
        v /= 3
        v /= v.norm()
        return v


# def alpha_weight(current_epoch, total_epoch):
#     af = 2
#     return (current_epoch + 1 / total_epoch) * af

def alpha_weight(current_epoch, total_epoch):
    af = 2
    # if current_epoch > total_epoch - 1:
    #     return af
    # else:
    return ((current_epoch + 1) / (total_epoch)) * af
```

vision_transformer.py

```
"""
Mostly copy-paste from timm library.
```

```
https://github.com/rwightman/pytorch-image-models/blob/master/timm/models/vision_transformer.py
"""
import math
from functools import partial

import torch
import torch.nn as nn

from utils import trunc_normal_


def drop_path(x, drop_prob: float = 0., training: bool = False):
    if drop_prob == 0. or not training:
        return x
    keep_prob = 1 - drop_prob
    shape = (x.shape[0],) + (1,) * (x.ndim - 1)  # work with diff dim tensors, not just 2D ConvNets
    random_tensor = keep_prob + torch.rand(shape, dtype=x.dtype, device=x.device)
    random_tensor.floor_()  # binarize
    output = x.div(keep_prob) * random_tensor
    return output


class DropPath(nn.Module):
    """Drop paths (Stochastic Depth) per sample  (when applied in main path of residual blocks).
```

```
    """
    def __init__(self, drop_prob=None):
        super(DropPath, self).__init__()
        self.drop_prob = drop_prob

    def forward(self, x):
        return drop_path(x, self.drop_prob, self.training)


class Mlp(nn.Module):
    def __init__(self, in_features, hidden_features=None, out_features=None,
act_layer=nn.GELU, drop=0.):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
```

```
        return x



class Attention(nn.Module):
    def __init__(self, dim, num_heads=8, qkv_bias=False, qk_scale=None, attn_drop=0., proj_drop=0.):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim ** -0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
```

```
        x = self.proj(x)
        x = self.proj_drop(x)
        return x, attn


class Block(nn.Module):
    def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False, qk_scale=None, drop=0., attn_drop=0.,
                 drop_path=0., act_layer=nn.GELU, norm_layer=nn.LayerNorm):
        super().__init__()
        self.norm1 = norm_layer(dim)
        self.attn = Attention(
            dim, num_heads=num_heads, qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=drop)
        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)

    def forward(self, x, return_attention=False):
        y, attn = self.attn(self.norm1(x))
        if return_attention:
            return attn
        x = x + self.drop_path(y)
        x = x + self.drop_path(self.mlp(self.norm2(x)))
        return x
```

```
class PatchEmbed(nn.Module):
    """ Image to Patch Embedding
    """
    def __init__(self, img_size=224, patch_size=16, in_chans=3, embed_dim=768):
        super().__init__()
        num_patches = (img_size // patch_size) * (img_size // patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.num_patches = num_patches

        self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size,
stride=patch_size)

    def forward(self, x):
        B, C, H, W = x.shape
        x = self.proj(x).flatten(2).transpose(1, 2)
        return x


class VisionTransformer(nn.Module):
    """ Vision Transformer """
    def __init__(self, img_size=[224], patch_size=16, in_chans=3, num_classes=0,
embed_dim=768, depth=12,
                 num_heads=12, mlp_ratio=4., qkv_bias=False, qk_scale=None, drop_rate=0.,
attn_drop_rate=0.,
```

```
        drop_path_rate=0., norm_layer=nn.LayerNorm, return_logits=False, **kwargs):
    super().__init__()
    self.num_features = self.embed_dim = embed_dim
    self.return_logits = return_logits

    self.patch_embed = PatchEmbed(
      img_size=img_size[0], patch_size=patch_size, in_chans=in_chans,
embed_dim=embed_dim)
    num_patches = self.patch_embed.num_patches

    self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
    self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim))
    self.pos_drop = nn.Dropout(p=drop_rate)

    dpr = [x.item() for x in torch.linspace(0, drop_path_rate, depth)]  # stochastic depth
decay rule
    self.blocks = nn.ModuleList([
      Block(
        dim=embed_dim, num_heads=num_heads, mlp_ratio=mlp_ratio,
qkv_bias=qkv_bias, qk_scale=qk_scale,
        drop=drop_rate, attn_drop=attn_drop_rate, drop_path=dpr[i],
norm_layer=norm_layer)
      for i in range(depth)])
    self.norm = norm_layer(embed_dim)

    # Classifier head
    self.head = nn.Linear(embed_dim, num_classes) if num_classes > 0 else nn.Identity()
```

```
    self.sigmoid = nn.Sigmoid()

    trunc_normal_(self.pos_embed, std=.02)
    trunc_normal_(self.cls_token, std=.02)
    self.apply(self._init_weights)

def _init_weights(self, m):
    if isinstance(m, nn.Linear):
        trunc_normal_(m.weight, std=.02)
        if isinstance(m, nn.Linear) and m.bias is not None:
            nn.init.constant_(m.bias, 0)
    elif isinstance(m, nn.LayerNorm):
        nn.init.constant_(m.bias, 0)
        nn.init.constant_(m.weight, 1.0)

def interpolate_pos_encoding(self, x, w, h):
    npatch = x.shape[1] - 1
    N = self.pos_embed.shape[1] - 1
    if npatch == N and w == h:
        return self.pos_embed
    class_pos_embed = self.pos_embed[:, 0]
    patch_pos_embed = self.pos_embed[:, 1:]
    dim = x.shape[-1]
    w0 = w // self.patch_embed.patch_size
    h0 = h // self.patch_embed.patch_size
    # we add a small number to avoid floating point error in the interpolation
```

```
        # see discussion at https://github.com/facebookresearch/dino/issues/8
        w0, h0 = w0 + 0.1, h0 + 0.1
        patch_pos_embed = nn.functional.interpolate(
            patch_pos_embed.reshape(1, int(math.sqrt(N)), int(math.sqrt(N)), dim).permute(0, 3, 1, 2),
            scale_factor=(w0 / math.sqrt(N), h0 / math.sqrt(N)),
            mode='bicubic',
        )
        assert int(w0) == patch_pos_embed.shape[-2] and int(h0) == patch_pos_embed.shape[-1]
        patch_pos_embed = patch_pos_embed.permute(0, 2, 3, 1).view(1, -1, dim)
        return torch.cat((class_pos_embed.unsqueeze(0), patch_pos_embed), dim=1)

    def prepare_tokens(self, x):
        B, nc, w, h = x.shape
        x = self.patch_embed(x)  # patch linear embedding

        # add the [CLS] token to the embed patch tokens
        cls_tokens = self.cls_token.expand(B, -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)

        # add positional encoding to each token
        x = x + self.interpolate_pos_encoding(x, w, h)

        return self.pos_drop(x)

    def forward(self, x):
```

```
    x = self.prepare_tokens(x)
    for blk in self.blocks:
        x = blk(x)
    x = self.norm(x)
    if self.return_logits == False:
        return x[:, 0]
    else:
        x = self.head(x[:, 0])
        x = self.sigmoid(x)
        return x

def get_last_selfattention(self, x):
    x = self.prepare_tokens(x)
    for i, blk in enumerate(self.blocks):
        if i < len(self.blocks) - 1:
            x = blk(x)
        else:
            # return attention of the last block
            return blk(x, return_attention=True)

def get_intermediate_layers(self, x, n=1):
    x = self.prepare_tokens(x)
    # we return the output tokens from the `n` last blocks
    output = []
    for i, blk in enumerate(self.blocks):
        x = blk(x)
```

```
            if len(self.blocks) - i <= n:
                output.append(self.norm(x))
        return output


def vit_tiny(patch_size=16, **kwargs):
    model = VisionTransformer(
        patch_size=patch_size, embed_dim=192, depth=12, num_heads=3, mlp_ratio=4,
        qkv_bias=True, norm_layer=partial(nn.LayerNorm, eps=1e-6), **kwargs)
    return model


def vit_single_head(patch_size=16, **kwargs):
    model = VisionTransformer(
        patch_size=patch_size, embed_dim=384, depth=6, num_heads=1, mlp_ratio=4,
        qkv_bias=True, norm_layer=partial(nn.LayerNorm, eps=1e-6), **kwargs)
    return model


def vit_small(patch_size=16, **kwargs):
    model = VisionTransformer(
        patch_size=patch_size, embed_dim=384, depth=12, num_heads=6, mlp_ratio=4,
        qkv_bias=True, norm_layer=partial(nn.LayerNorm, eps=1e-6), **kwargs)
    return model
```

```
def vit_base(patch_size=16, **kwargs):
    model = VisionTransformer(
        patch_size=patch_size, embed_dim=768, depth=12, num_heads=12, mlp_ratio=4,
        qkv_bias=True, norm_layer=partial(nn.LayerNorm, eps=1e-6), **kwargs)
    return model



class DINOHead(nn.Module):
    def __init__(self, in_dim, out_dim, use_bn=False, norm_last_layer=True, nlayers=3,
hidden_dim=2048, bottleneck_dim=256):
        super().__init__()
        nlayers = max(nlayers, 1)
        if nlayers == 1:
            self.mlp = nn.Linear(in_dim, bottleneck_dim)
        else:
            layers = [nn.Linear(in_dim, hidden_dim)]
            if use_bn:
                layers.append(nn.BatchNorm1d(hidden_dim))
            layers.append(nn.GELU())
            for _ in range(nlayers - 2):
                layers.append(nn.Linear(hidden_dim, hidden_dim))
                if use_bn:
                    layers.append(nn.BatchNorm1d(hidden_dim))
                layers.append(nn.GELU())
            layers.append(nn.Linear(hidden_dim, bottleneck_dim))
            self.mlp = nn.Sequential(*layers)
```

```
        self.apply(self._init_weights)
        self.last_layer = nn.utils.weight_norm(nn.Linear(bottleneck_dim, out_dim, bias=False))
        self.last_layer.weight_g.data.fill_(1)
        if norm_last_layer:
            self.last_layer.weight_g.requires_grad = False

    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            trunc_normal_(m.weight, std=.02)
            if isinstance(m, nn.Linear) and m.bias is not None:
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.mlp(x)
        x = nn.functional.normalize(x, dim=-1, p=2)
        x = self.last_layer(x)
        return x



class CLS_head(nn.Module):
    def __init__(self, in_dim, hidden_dim, num_class):
        super().__init__()
        self.mlp = nn.Sequential(nn.Linear(in_dim, hidden_dim),
                                 nn.ReLU(),
                                 nn.Linear(hidden_dim, hidden_dim),
                                 nn.ReLU(),
```

```
                    nn.Linear(hidden_dim, num_class))
  def forward(self, x):
    x = self.mlp(x)
    return x
```

visualize_attention.py

```
import os
import argparse
import cv2
import random
import colorsys

import skimage.io
from skimage.measure import find_contours
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms as pth_transforms
import numpy as np
from PIL import Image

import utils
```

```
import vision_transformer as vit_o
import glob


def apply_mask(image, mask, color, alpha=0.5):
    for c in range(3):
        image[:, :, c] = image[:, :, c] * (1 - alpha * mask) + alpha * mask * color[c] * 255
    return image


def random_colors(N, bright=True):
    """
    Generate random colors.
    """
    brightness = 1.0 if bright else 0.7
    hsv = [(i / N, 1, brightness) for i in range(N)]
    colors = list(map(lambda c: colorsys.hsv_to_rgb(*c), hsv))
    random.shuffle(colors)
    return colors


def display_instances(image, mask, fname="test", figsize=(5, 5), blur=False, contour=True,
alpha=0.5):
    fig = plt.figure(figsize=figsize, frameon=False)
    ax = plt.Axes(fig, [0., 0., 1., 1.])
    ax.set_axis_off()
    fig.add_axes(ax)
    ax = plt.gca()
```

```
N = 1
mask = mask[None, :, :]
# Generate random colors
colors = random_colors(N)

# Show area outside save boundaries.
height, width = image.shape[:2]
margin = 0
ax.set_ylim(height + margin, -margin)
ax.set_xlim(-margin, width + margin)
ax.axis('off')
masked_image = image.astype(np.uint32).copy()
for i in range(N):
  color = colors[i]
  _mask = mask[i]
  if blur:
    _mask = cv2.blur(_mask,(10,10))
  # Mask
  masked_image = apply_mask(masked_image, _mask, color, alpha)
  # Mask Polygon
  # Pad to ensure proper polygons for masks that touch save edges.
  if contour:
    padded_mask = np.zeros((_mask.shape[0] + 2, _mask.shape[1] + 2))
    padded_mask[1:-1, 1:-1] = _mask
    contours = find_contours(padded_mask, 0.5)
```

```
        for verts in contours:
            # Subtract the padding and flip (y, x) to (x, y)
            verts = np.fliplr(verts) - 1
            p = Polygon(verts, facecolor="none", edgecolor=color)
            ax.add_patch(p)
    ax.imshow(masked_image.astype(np.uint8), aspect='auto')
    fig.savefig(fname)
    print(f"{fname} saved.")
    return


if __name__ == '__main__':
    parser = argparse.ArgumentParser('Visualize Self-Attention maps')
    parser.add_argument('--arch', default='vit_small', type=str,
        choices=['vit_tiny', 'vit_small', 'vit_base'], help='Architecture (support only ViT atm).')
    parser.add_argument('--patch_size', default=8, type=int, help='Patch resolution of the model.')
    parser.add_argument('--pretrained_weights', default='/4TB_hdd/TB_checkpoint/20220209_Experiments/20220209_fold_3/checkpoint.pth', type=str,
        help="Path to pretrained weights to load.")
    parser.add_argument("--checkpoint_key", default="student", type=str,
        help='Key to use in the checkpoint (example: "student")')
    parser.add_argument("--image_dir", default="/COVID_4TB/open_3folds/test/Tuberculosis/", type=str, help="Path of png images to load.")
    parser.add_argument("--image_size", default=(256, 256), type=int, nargs="+", help="Resize save.")
```

```
    parser.add_argument('--output_dir', default='./temp_attention/', help='Path where to save visualizations.')
    parser.add_argument("--threshold", type=float, default=None, help="""We visualize masks
        obtained by thresholding the self-attention maps to keep xx% of the mass.""")
    args = parser.parse_args()

    device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
    # build model
    model = vit_o.vit_single_head(patch_size=8)
    for p in model.parameters():
        p.requires_grad = False
    model.eval()
    model.to(device)

    if os.path.isfile(args.pretrained_weights):
        state_dict = torch.load(args.pretrained_weights, map_location="cpu")
        if args.checkpoint_key is not None and args.checkpoint_key in state_dict:
            print(f"Take key {args.checkpoint_key} in provided checkpoint dict")
            state_dict = state_dict[args.checkpoint_key]
        # remove `module.` prefix
        state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
        # remove `backbone.` prefix induced by multicrop wrapper
        state_dict = {k.replace(".backbone.", "___"): v for k, v in state_dict.items()}
        state_dict = {k.replace("backbone.", ""): v for k, v in state_dict.items()}
        state_dict = {k.replace("___", ".backbone."): v for k, v in state_dict.items()}
        msg = model.load_state_dict(state_dict, strict=False)
```

```
    print('Pretrained weights found at {} and loaded with msg: {}'.format(args.pretrained_weights, msg))

  else:
    print("Please use the `--pretrained_weights` argument to indicate the path of the checkpoint to evaluate.")
    url = None
    if args.arch == "vit_small" and args.patch_size == 16:
      url = "dino_deitsmall16_pretrain/dino_deitsmall16_pretrain.pth"
    elif args.arch == "vit_small" and args.patch_size == 8:
      url = "dino_deitsmall8_300ep_pretrain/dino_deitsmall8_300ep_pretrain.pth"  # model used for visualizations in our paper
    elif args.arch == "vit_base" and args.patch_size == 16:
      url = "dino_vitbase16_pretrain/dino_vitbase16_pretrain.pth"
    elif args.arch == "vit_base" and args.patch_size == 8:
      url = "dino_vitbase8_pretrain/dino_vitbase8_pretrain.pth"
    if url is not None:
      print("Since no pretrained weights have been provided, we load the reference pretrained DINO weights.")
      state_dict = torch.hub.load_state_dict_from_url(url="https://dl.fbaipublicfiles.com/dino/" + url)
      model.load_state_dict(state_dict, strict=True)
    else:
      print("There is no reference weights available for this model => We use random weights.")

  img_dir = args.image_dir
  images = glob.glob(img_dir + '/**/*.png', recursive=True)
```

```
  for img_path in images:
    image = cv2.imread(img_path, 1)
    image = cv2.resize(image, dsize=(256, 256), interpolation=cv2.INTER_LINEAR)
    image = Image.fromarray(image)

    img = pth_transforms.Compose(
      [utils.GaussianBlurInference(),
       pth_transforms.ToTensor()])(image)

    # make the save divisible by the patch size
    w, h = img.shape[1] - img.shape[1] % args.patch_size, img.shape[2] - img.shape[2] % args.patch_size
    img = img[:, :w, :h].unsqueeze(0)

    w_featmap = img.shape[-2] // args.patch_size
    h_featmap = img.shape[-1] // args.patch_size

    attentions = model.get_last_selfattention(img.to(device))

    nh = attentions.shape[1] # number of head

    # we keep only the output patch attention
    attentions = attentions[0, :, 0, 1:].reshape(nh, -1)

    if args.threshold is not None:
```

```
        # we keep only a certain percentage of the mass
        val, idx = torch.sort(attentions)
        val /= torch.sum(val, dim=1, keepdim=True)
        cumval = torch.cumsum(val, dim=1)
        th_attn = cumval > (1 - args.threshold)
        idx2 = torch.argsort(idx)
        for head in range(nh):
            th_attn[head] = th_attn[head][idx2[head]]
        th_attn = th_attn.reshape(nh, w_featmap, h_featmap).float()
        # interpolate
        th_attn = nn.functional.interpolate(th_attn.unsqueeze(0),
scale_factor=args.patch_size, mode="nearest")[0].cpu().numpy()

    attentions = attentions.reshape(nh, w_featmap, h_featmap)
    attentions = nn.functional.interpolate(attentions.unsqueeze(0),
scale_factor=args.patch_size, mode="bilinear")[0].cpu().numpy()

    # save attentions heatmaps
    output_dir = args.output_dir + img_path.split('/')[-1] + '/'
    os.makedirs(output_dir, exist_ok=True)
    torchvision.utils.save_image(torchvision.utils.make_grid(img, normalize=True,
scale_each=True), os.path.join(output_dir, "img.png"))
    for j in range(nh):
        fname = os.path.join(output_dir, "attn-head" + str(j) + ".png")
        attentions[j] = (attentions[j] - attentions[j].min()) / (attentions[j].max() -
attentions[j].min())
```

```
        plt.imsave(fname=fname, arr=attentions[j], format='png', cmap='gray')
        print(f"{fname} saved.")

        if args.threshold is not None:
            image = skimage.io.imread(os.path.join(output_dir, "img.png"))
            for j in range(nh):
                display_instances(image, th_attn[j], fname=os.path.join(args.output_dir,
"mask_th" + str(args.threshold) + "_head" + str(j) +".png"), blur=False)
```